4

RADC-TR-89-162
Final Technical Report
October 1989

ALGORITHMS FOR FAULT TOLERANT DISTRIBUTED SYSTEMS

SRI International

Leslie Lamport, P. Michael Melliar-Smith, Louise Moser, Ira Greenberg, John Rushby

AD-A214 447

This effort was funded totally by the Laboratory Director's fund.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, NY 13441-5700

89 11 15 016

RADC-TR-89-162 has been reviewed and is approved for publication.

APPROVED: [illegible]

THOMAS F. LAWRENCE
Project Engineer

APPROVED: [illegible]

RAYMOND P. URTZ, Jr.
Technical Director
Directorate of Command & Control

FOR THE COMMANDER: [illegible]

IGOR G. PLONISCH
Directorate of Plans & Programs

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE	Form Approved OMB No. 0704-0188
1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS N/A
2a. SECURITY CLASSIFICATION AUTHORITY N/A	3. DISTRIBUTION/AVAILABILITY OF REPORT Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
4. PERFORMING ORGANIZATION REPORT NUMBER(S) N/A	5. MONITORING ORGANIZATION REPORT NUMBER(S) RADC-TR-89-162
6a. NAME OF PERFORMING ORGANIZATION SRI International; 6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Rome Air Development Center (COTD)
6c. ADDRESS (City, State, and ZIP Code) Computer Science Laboratory 333 Ravenswood Ave Menlo Park CA 94025	7b. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700
8a. NAME OF FUNDING/SPONSORING ORGANIZATION Rome Air Development Center; 8b. OFFICE SYMBOL (If applicable) COTD	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER F30602-85-C-0024
8c. ADDRESS (City, State, and ZIP Code) Griffiss AFB NY 13441-5700	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 61101F; PROJECT NO. LDFP; TASK NO 06; WORK UNIT ACCESSION NO. C4

11. TITLE (Include Security Classification)
ALGORITHMS FOR FAULT TOLERANT DISTRIBUTED SYSTEMS

12. PERSONAL AUTHOR(S)
Leslie Lamport, P. Michael Melliar-Smith, Louise Moser, Ira Greenberg, and John Rushby

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Final	FROM Feb 85 TO Feb 88	October 1989	214

16. SUPPLEMENTARY NOTATION
This effort was funded totally by the Laboratory Director's fund.

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Interprocess Communication, Broadcast Protocols, Interval Logic, Fault Tolerance, Multiprocessor
12	07		

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

The research described in this report is presented in six parts: (1) On Interprocess Communication studies interprocess communication without assuming any lower-level communication primitives. A formalism is developed for reasoning about concurrent systems that does not assume an atomic grain of action, (2) The Intersecting Broadcast Machine is a novel array processor architecture, capable of processing efficiently programs whose arbitrary or complex structure would make them difficult to map onto conventional array processors. The architecture also supports fault-tolerant operation, (3) Broadcast Protocols for Distributed Systems considers how the broadcast character of communications media such as Ethernet and packet radio can be exploited to yield reliable communication with very little overhead, (4) Extending Interval Logic to Real Time Systems presents a technique for the formal expression of the real-time constraints that are critical to the specification of fault-tolerant distributed systems, (5) Consistency of Replicated Information in Multichannel Fault Tolerant Systems considers the possibility of using (over)

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT ☐ UNCLASSIFIED/UNLIMITED [✓ SAME AS RPT.] ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL Thomas F. Lawrence	22b. TELEPHONE (Include Area Code) (315) 330-2158; 22c. OFFICE SYMBOL RADC (COTD)

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED

UNCLASSIFIED

Block 19 (Continued)

similar, but not identical, processing in the replicas of a fault-tolerant system. Conventional fault-tolerant systems using replicated processing require the replicas to be identical, so that they can be compared by exact match algorithms. This exact replication increases the risk that a common fault will affect all replicas and cause system failure, and (6) Experimental Implementation and Evaluation of the TRANS Broadcast Protocol describes an implementation and evaluation of the broadcast protocol outlined in Part III.

UNCLASSIFIED

TABLE OF CONTENTS

TABLE OF FIGURES

LIST OF TABLES

The research described in this report is presented in six parts.

Part I: *On Interprocess Communication* studies interprocess communication without assuming any lower-level communication primitives. A formalism is developed for reasoning about concurrent systems that does not assume an atomic grain of action.

Part II: *The Intersecting Broadcast Machine* is a novel array processor architecture, capable of processing efficiently programs whose arbitrary or complex structure would make them difficult to map onto conventional array processors. The architecture also supports fault-tolerant operation.

Part III: *Broadcast Protocols for Distributed Systems* considers how the broadcast character of communications media such as Ethernet and packet radio can be exploited to yield reliable communication with very little overhead.

Part IV: *Extending Interval Logic to Real Time Systems* presents a technique for the formal expression of the real-time constraints that are critical to the specification of fault-tolerant distributed systems.

Part V: *Consistency of Replicated Information in Multichannel Fault Tolerant Systems* considers the possibility of using similar, but not identical, processing in the replicas of a fault tolerant system. Conventional fault tolerant systems using replicated processing require the replicas to be identical, so that they can be compared by exact match algorithms. This exact replication increases the risk that a common fault will affect all replicas and cause system failure.

Part VI: *Experimental Implementation and Evaluation of the* TRANS *Broadcast Protocol* describes an implementation and evaluation of the broadcast protocol outlined in Part III.

Part I

On Interprocess Communication

1.1 On Interprocess Communication

All communication ultimately involves a communication medium whose state is changed by the sender and observed by the receiver. A sending processor changes the voltage on a wire and a receiving processor observes the voltage change; a speaker changes the vibrational state of the air and a listener senses this change.

Communication acts can be divided into two classes: *transient* and *persistent*. In a transient communication, the medium's state is changed only for the duration of the communication, immediately afterwords reverting to its "normal" state. A message sent on an ethernet modifies the transmission medium's state only while the message is in transit; the altered state of the air lasts only while the speaker is talking. In a persistent communication, the state change remains after the sender has finished its communication. Setting a voltage level on a wire, writing on a blackboard, and raising a flag on a flagpole are all examples of persistent communication.

Transient communication is possible only if the receiver is observing the communication medium while the sender is modifying it. This implies an *a priori* synchronization—the receiver must be waiting for the communication to take place. Communication between truly asynchronous processes must be persistent, the sender changing the state of the medium and the receiver able to sense that change at a later time.

Message passing is often considered to be a form of transient communication between asynchronous processes. However, a closer examination of asynchronous message passing reveals that it involves a persistent communication. Messages are placed in a buffer that is periodically tested by the receiver. Viewed at a low level, message passing is typically accomplished by putting a message in a buffer and setting an interrupt bit that is tested on every machine instruction. The receiving process actually consists of two asynchronous subprocesses: a *main* process that is usually thought of as the receiver, and an *input* process that continuously monitors the communication medium and puts messages in the buffer. The input process is synchronized with the sender (it is a "slave" process) and communicates asynchronously with the main process using the buffer as a medium for persistent communication.

The subject of this report is asynchronous interprocess communication, so only persistent communication is considered. Moreover, we will restrict ourselves to unidirectional communication, in which only a single process can modify the state of the medium. With this restriction, two-way communication requires at least two separate communication media, one modified by each process. However, multiple receivers will be considered. We also restrict our attention to discrete systems, in which the medium has a finite number of distinguishable states. The sender can therefore set the medium to one of a fixed number of persistent states, and the receiver(s) can observe the medium's state.

The form of persistent communication that we have described is more commonly known as a shared register, where the sender and receiver are called the *writer* and *reader*, respectively, and the state of the communication medium is known as the *value* of the register. We will use these in the rest of this paper, so we will consider finite-valued registers with a single writer and one or more readers.

While the practical applications of the algorithms described in this paper will be to "small" registers, the larger purpose is to develop insight into, and formal methods for reasoning about, nonatomic operations to data objects. In the realm of conventional database theory, atomicity is usually called "serializability". Moreover, although the notation used in describing the algorithms suggests a shared-memory implementation, these are really distributed algorithms, since each shared register is modified by only a single process. Thus, the results described here can be regarded as a preliminary investigation of nonserializable operations in a distributed database.

In assuming a single writer, we rule out the possibility of concurrent writes (to the same register). Since a reader only senses the value, there is no reason why a read operation must interfere with another read or write operation. (While reads do interfere with other operations in some forms of memory, such as magnetic core, this interference is an idiosynchracy of the particular technology rather than an inherent property of reading.) We therefore assume that a read does not affect any other read or any write. However, it is not clear what effect a concurrent write should have on a read.

In concurrent programming, one traditionally assumes that a writer has exclusive access to shared data, making concurrent reading and writing impossible. This assumption is enforced either by requiring the programming language to provide the necessary exclusive access, or by implementing the exclusion with a "readers-writers" protocol [3]. Such an approach requires that a reader must wait while a writer is accessing the register, and vice-versa. Moreover, any method for achieving such exclusive access, whether implemented by the programmer or the compiler, requires a lower-level shared register. At some level, the problem of concurrent access to a shared register must be faced. It is this problem that will be addressed, so we eschew any approach that requires one process to wait for another.

Asynchronous concurrent access to shared registers is usually considered only at the hardware level, so it is at this level that the methods developed here could have some direct application. However, concurrent access to shared data occurs at high levels of abstraction. One cannot allow any single process exclusive access to the entire social security system's database. While algorithms for implementing a single register cannot be applied to such a database, we hope that the formalism developed for analyzing these algorithms will eventually prove useful for analyzing concurrent systems at a higher-level. Nevertheless, it is probably best to think of a register as a low-level component, probably implemented in hardware, when reading this paper.

Hardware implementations of asynchronous communication often make assumptions about the relative speeds of the communicating processes. Such assumptions can lead to simplifications. For example, the problem of constructing an atomic register, discussed below, is shown to be easily solved by assuming that two successive reads of a register cannot be concurrent with a single write. If one knows how long a write can take, a delay can be added between successive reads to ensure that this assumption holds. We make no such assumptions about process speeds. The results therefore apply even to communication between processes of vastly differing speeds.

We therefore make no assumptions about relative process speed and consider a shared register in which a read can overlap (be concurrent with) a write. Three possible assumptions about what can happen when a read overlaps one or more writes are considered.

The weakest possibility is a *safe* register, in which the only assumption made about the value obtained by a read that overlaps a write is that the read obtain one of the possible values of the register—for example, a read of a boolean-valued register must obtain either *true* or *false*. A read that is not concurrent with a write is assumed to obtain the correct value—that is, the most recently written one. However, a read that overlaps a write may return any possible value.

The next stronger possibility is a *regular* register, which is safe (a read not concurrent with a write gets the correct value) and in which a read that overlaps a write obtains either the old or new value. More generally, a read that overlaps any series of writes obtains either the value before the first of the writes or one of the values being written.

The final possibility is an *atomic* register, which is safe and in which reads and writes behave as if they occurred in some definite order. In other words, for any execution of the system, there is some way of totally ordering the reads and writes so that the values returned by the reads are the same as if the operations had been performed in that order, with no overlapping. (It is also required that this ordering should be a reasonable one; the precise condition is stated below.)

A regular register is obviously stronger than a safe one, since it places a condition on the value returned by a read that overlaps a write. An atomic register is stronger than a regular one because, if two successive reads overlap the same write, then a regular register allows the first read to obtain the new value and the second read the old value. This is forbidden in an atomic register, in which the only allowed possibilities are old-old, new-new, and old-new. In fact, it will be shown that a regular register is atomic if and only if two successive reads that overlap the same write cannot obtain the new then the old value. Thus, a regular register is automatically an atomic one if two successive reads cannot overlap the same write.

These are the only three general classes of register that we have been able to think of. Each class merits study. Safety seems to be the weakest requirement that allows useful communication; we do not know how to achieve any form of interprocess synchronization with a weaker assumption. Regularity asserts that a read returns a "reasonable" value, and seems to be a natural requirement. Atomicity is the most common assumption made about shared registers, and is provided by current multiport computer memories.[1] At a lower level, such as interprocess communication within a single chip, only safe registers are provided; other classes of register must be implemented using safe ones.

Any method of implementing a single-writer register can be classified by three "coordinates" with the following values:

- *safe*, *regular*, or *atomic*, according to the strongest assumption that the register satisfies.
- *boolean* or *multivalued*, according to whether the method produces only boolean registers or registers with any desired number of values.
- *single-reader* or *multireader*, according to whether the method yields registers with only one reader or with any desired number of readers.

This produces twelve classes of implementations, partially ordered by "strength"—for example, a method that produces atomic, multivalued, multireader registers is stronger than one producing regular, multivalued, single-reader registers. In this paper, we address the problem of implementing a register of one class using one or more registers of a weaker class.

The weakest class of register, and therefore the easiest to implement, is a safe, boolean, single-reader one. This seems to be the most natural kind of register to implement with current hardware technology, requiring only that the writer set a voltage level either high or low and that the reader test this level without disturbing it. A series of constructions of stronger registers from weaker ones is presented that allows almost every class of register to be constructed starting from this weakest class. The one exception is that constructing an atomic, multireader register from any weaker one is still an open problem. Most of the constructions are simple; the difficult ones are Construction 4 that implements an m-reader multivalued regular register using m-reader boolean regular registers, and Construction 5 that

[1]However, the standard implementation of a multiport memory does not meet our requirements for an asynchronous register because, if two processes concurrently access a memory cell, one must wait for the other.

implements a single-reader multivalued atomic register using single-reader multivalued regular registers.

We have defined three classes of shared registers for asynchronous interprocess communication, and provided algorithms for implementing one class in terms of a weaker class. For single-writer registers, the only unsolved problem is implementing a multi-reader atomic register. A solution probably exists, but it undoubtedly requires that a reader communicate with all other readers as well as with the writer. Also, more efficient implementations than Constructions 4 and 5 probably exist. For multivalued registters, Peterson's algorithm [11] combined with Construction 5 provides a more efficient implementation of a regular register than Construction 4, and a more efficient implementation of a single-reader atomic register than Construction 5. However, in this solution, Construction 4 is still needed to implement the regular register used in Construction 5.

We have not addressed the question of multi-writer shared registers. It is not clear what assumptions one should make about the effect of overlapping writes. The one case that is straightforward is that of an atomic multi-writer register—the kind of register traditionally assumed in shared-variable concurrent programs. This raises the problem of implementing a multi-writer atomic register from single-writer ones. An unpublished algorithm of Bard Bloom implements a two-writer atomic register using single-writer atomic registers.

In addition to studying shared registers, we have also developed a formalism for reasoning about concurrent systems that is not based upon atomic actions. Starting from a more general, relativistic viewpoint, we showed that one can, with no essential loss of generality, think in terms of starting and finishing times of operations. While starting and finishing times are intuitively more appealing, and can be useful in proving metatheorems about general systems, rigorous reasoning about specific algorithms is best done in the general formalism, using Axioms A1–A5. These axioms seem to contain the fundamental properties of temporal relations among operation executions that are needed to analyze concurrent algorithms.

1.2 The Constructions

In this section, the algorithms for constructing different classes of registers are described and informally justified. Rigorous correctness proofs are postponed until Section 1.4, after the necessary formalism is developed.

The algorithms are described by indicating how a write and a read are performed. I will not bother to indicate the initial state of the shared registers—it is the one that would result from writing the initial value starting from any arbitrary state.

The first construction implements a multireader safe or regular register from single-reader ones. It uses the obvious method of having the writer simply maintain a separate copy of the register for each reader. The **for all** statement denotes that its body is executed once for each of the indicated values of i; these separate executions can be done in any order or concurrently.

Construction 1 *Let $v_1, \ldots, v_m$ be single-reader, n-valued registers, where each v_i can be written by the same writer and read by process i, and construct a single n-valued register v in which the operation $v := \mu$ is performed as follows:*

```
for all $i$ in $\{1, \ldots, m\}$
    do $v_i := \mu$ od
```

and process i reads v by reading the value of v_i. If the v_i are safe or regular registers, then v is a safe or regular register, respectively.

Any read by process i that does not overlap a write of v does not overlap a write of v_i. If v_i is safe, then this read gets the correct value, which shows that v is safe. If a read of v_i by process i overlaps a write of v_i, then it overlaps the write of the same value to v. It follows easily from this that if v_i is regular, then v is also regular.

This construction does not make v an atomic register even if the v_i are atomic. If reads by two different processes i and j both overlap the same write, it is possible for i to get the new value and j the old value even though the read by i precedes the read by j—a possibility no allowed by an atomic register.

The next construction is also trivial; it implements an n-bit safe register from n single-bit ones.

Construction 2 *Let $v_1, \ldots, v_n$ be boolean m-reader registers, each written by the same writer and read by the same set of readers. Let v be the 2^n-valued, m-reader register in which the number with binary representation $\mu_1 \ldots \mu_n$ is written by*

```
for all $i$ in $\{1, \ldots, m\}$ do $v_i := \mu_i$ od
```

and in which the value is read by reading all the v_i. If each v_i is safe, then v is safe.

The register v is not regular even if the v_i are. A read can return any value if it overlaps a write that changes the register's value from $0 \ldots 0$ to $1 \ldots 1$.

The next construction shows that it is trivial to implement a boolean regular register from a safe boolean register. In a safe register, a read that overlaps a write may get any value, while in a regular register it must get either the old or new value. However, a read of a safe boolean register must obtain either *true* or *false* on any read, so it must return either the old or new value if it overlaps a write that changes the value. A boolean safe register can fail to be regular only if a read that overlaps a write that does not change the value returns the other (wrong) value. To prevent this possibility, one simply does not perform a write that does not change the value.

Construction 3 *Let v be an m-reader boolean register, and let x be a variable internal to the writer (not a shared register) initially equal to the initial value of v. Define v^* to be the m-reader boolean register in which the write operation $v^* := \mu$ is performed as follows:*

```
if $x \neq \mu$ then $v := \mu$;
                     $x := \mu$
fi
```

and a read of v^* is performed by reading v. If v is safe then v^* is regular.

There are two known algorithms for implementing a multivalued regular register from boolean ones. The simpler one employs a unary encoding, in which the value μ is denoted by zeros in bits 0 through $\mu - 1$ and a one in bit μ. A reader reads the bits from left to right (0 to n) until it finds a one. To write the value μ, the writer first sets v_μ to one and then sets bits $\mu - 1$ through 1 to zero, writing from right to left. (The idea of implementing shared data by reading and writing its components in different directions was also used in [4].)

Construction 4 *Let $v_1, \ldots, v_n$ be boolean, m-reader registers, and let v be the n-valued, m-reader register in which the operation $v := \mu$ is performed by*

```
$v_\mu := 1$;
**for** $i := \mu - 1$ **step** $-1$ **until** 1 **do** $v_i := 0$ **od**
```

and a read is performed by:

```
$\mu := 1$;
**while** $v_\mu = 0$ **do** $\mu := \mu + 1$ **od**;
*return* $\mu$
```

If each v_i is regular, then v is regular.

The correctness of this algorithm is not at all obvious. Indeed, it is not even obvious that the **while** loop in the read operation does not "fall off the end" and try to read the nonexistent register v_{n+1}. This can't happen because whenever the writer writes a zero, there is a one to the right of it. (Since I am assuming that an initial value has been written, some v_i initially equals one.) As an exercise, the reader of this paper can convince himself that whenever a reading process sees a one, it was written by either a concurrent write or by the most recent preceding one, so v is regular. The formal proof is given in Section 1.4.

The value of v_n is only set to one, never to zero. It can therefore be eliminated; the writer simply never writes it and the reader assumes its value is one instead of reading it. I will not bother writing down this modification.

Even if all the v_i are atomic, Construction 4 does not produce an atomic register. To see this, suppose that the register initially has the value 3, so $v_1 = v_2 = 0$ and $v_3 = 1$, the writer first writes the value 1 then the value 2, and there are two successive read operations. This can produce the following sequence of actions:

- the first read finds $v_1 = 0$
- the first write sets $v_1 := 1$
- the second write sets $v_2 := 1$
- the first read finds $v_2 = 1$ and returns the value 2
- the second read finds $v_1 = 1$ and returns the value 1.

In this scenario, the first read obtains a newer value (the one written by the second write) than the second read (which obtains the one written by the first write), even though it precedes the second read. This shows that the register is not atomic.

Construction 4 uses $n - 1$ boolean regular registers to make an n-valued one, so it is practical only for small values of n. We would like an algorithm that requires $O(\log n)$ boolean registers to construct an n-valued register. The second method for constructing a regular multivalued register uses an algorithm of Peterson [11] that implements an m-reader n-valued atomic register with $m + 2$ safe m-reader registers; $2m$ atomic boolean 2-reader registers, and two atomic boolean m-reader registers. There is no known algorithm for constructing multivalued m-reader atomic registers from simpler ones. However, we can apply Peterson's algorithm to construct an n-valued single-reader atomic register using three safe single-reader n-valued registers and four single-reader atomic boolean registers. The safe registers can be implemented with Construction 2, and the atomic boolean registers can be implemented with Construction 5 below. Since an atomic register is regular, Construction 1 can then be used to make an m-reader n-valued regular register from $O(3m \log n)$ single-reader boolean regular registers.

Before giving the algorithm for constructing a two-reader atomic register, I prove a result that indicates why no trivial algorithm will work. It asserts that there can be no algorithm in which the writer only writes and the reader only reads; any algorithm must involve two-way communication between the reader and the writer.

Theorem: *There exists no algorithm to implement an atomic register using only a finite number of regular registers that can be written by the writer (of the atomic register).*

Proof: I assume such an algorithm and derive a contradiction. Without loss of generality, I can assume that there is only a single regular register v written by the writer and read by the reader. (Any algorithm that works with multiple registers must also work when those registers are combined into a single large regular register.)

Let v^* denote the atomic register that is being implemented. Suppose that the writer performs an infinite number of writes that change the value of v^*. There must be some pair of values assumed by v^*, call them 0 and 1, such that there are an infinite number of writes that change v^*'s value from 0 to 1. Since v can assume only a finite number of values (the hypothesis states that the original algorithm has only a finite number of registers, and all registers are taken to have only a finite number of possible values), there must exist values $v_0, \ldots, v_n$ of v such that v_0 is the final value of v after each one of an infinite number of writes of 0 to v^*, v_n is the final value of v after each one of an infinite number of writes of 1 to v^*, and, for each $i < n$, the value of v is changed from v_i to v_{i+1} during infinitely many writes that change the value of v^* from 0 to 1.

A read of v^* may involve several reads of v. However, by considering only scenarios in which each of those reads of v obtains the same value, we may assume that each read of v^* reads v only once. Since v assumes each value v_i infinitely often, it must be possible for a sequence of $n+1$ consecutive reads to obtain the values $v_n, v_{n-1}, \ldots, v_1$.

The read that finds v equal to v_i and the subsequent read that finds v equal to v_{i-1} could both have overlapped the same write of v, which could have been a write that occured in the process of changing v^*'s value from 0 to 1. Therefore, if the read of v^* that finds v equal to v_i returns the value 1, then the subsequent read that finds v equal to v_{i-1} must also return the value 1, since both reads could be overlapping the same write and, in that case, two successive reads of an atomic register cannot return first the new then the old value.

The first read, which finds v equal to v_n, must return the value 1, since it could have occurred after the completion of a write of 1. By induction, this implies that the last read, which found v equal to v_0, must return the value 1. However, this read could have occurred after a write of 0 and before any subsequent write, so returning the value 1 would violate the assumption that the register v^* is safe. (An atomic register is *a fortiori* safe.) This is the required contradiction. ∎

This theorem could be expressed and proved using the formalism developed below, but doing so would lead to no new insight. The formal proof of this theorem is therefore left as an exercise for the compulsive reader.

The theorem is false if no bound is placed on the number of values a register can hold. Given a regular register v that can assume an unbounded number of values, an atomic register v^* is implemented as follows. The writer sets v equal to a pair consisting of the value of v^* and a sequential version number. The reader reads v and compares the version number with the previous one it read. If the new version number is higher, then it uses the value it just read; if the new version number is lower, then it forgets the value and version number it just read and uses the previously-read value. The correctness of this algorithm follows easily from Proposition 9 of Section 1.3.3. By assuming registers hold only a bounded set of values, I am disallowing such algorithms.

Finally, we come to the algorithm for constructing a single-reader atomic register from regular ones. To begin, we try to implement an atomic register v^* with a regular register v that holds a pair of values, both normally equal. When v is changed from (ν, ν) (denoting $v^* = \nu$) to (μ, μ) (denoting $v^* = \mu$), it is first set to the intermediate value (ν, μ). The reader reads v and returns the first component unless it obtains (ν, μ) after having returned the value μ the last time, in which case it must return the value μ to avoid a "new-old" sequence.

The preceding theorem shows that this idea, by itself, is not enough. The reader is in a quandary if three successive reads of v obtain the values (μ, μ), (ν, μ), and (ν, ν). The first read simply returns μ; as I just observed, the second read must also return μ; but what can the third read return? The second and third reads could both have overlapped a single write that is changing the value from ν to μ, so returning ν would produce a new-old sequence. On the other hand, the third read could have seen a completely new value, written long after the write that overlapped the second read, so returning μ could violate safety—the requirement that a read not overlapping any write return the correct value.

To overcome this problem, I add another bit to v, which I will call the *color* value. When the reader reads v, it sets a shared one-bit register cr to v's color value. The writer first reads the register cr and sets v to the opposite color. (Thus, the reader tries to make cr and v's color the same, and the writer tries to make them different.) The reader interprets (ν, μ) as a μ only if its previous read saw a μ of the same color. The only source of embarrassment is now if three successive reads return values (μ, μ), (ν, μ), and (ν, ν) that are all the same color. It will be shown in Section 4 that this can happen only if the last read actually overlaps the write of (ν, μ), so it is allowed to return the value μ without violating the safety requirement.

In the following construction, the variable cr is written by the reader and read by both the reader and the writer. A two-reader register is not needed, since the reader can maintain a local variable containing the value that it last wrote into cr. (This is just Construction 1 with $m = 2$ and the writer being the second reader.) Such a local variable would complicate the description, so it is omitted. In the reader's program, the primed variables denote the values read the previous time, except that if the reader reads (μ, μ) then (ν, μ), both with the same color, then it "forgets about" the latter value.

Construction 5 *Let $\mathcal{V}$ be an n-element set; let w and r be processes; let v, cw denote a single $2n^2$-valued register that can be written by w and read by r, where v has a value in $\mathcal{V} \times \mathcal{V}$ and cw is boolean valued; and let cr be a boolean register that can be written by r and read by w. Define the n-valued register v^*, with values in $\mathcal{V}$, written by w and read by r by letting the write $v^* := \mu$ be performed by:*

```
    $v, cw := (v_1, \mu), \neg cr;$
    $v, cw := (\mu, \mu), cw$
```

and letting the read operation be performed by the program of Figure 1.1, where x and x' are local variables in $\mathcal{V} \times \mathcal{V}$, cr' is a boolean-valued local variable, and rtn is a local variable with values in $\mathcal{V}$ whose final value is the one returned by the read. Initially, x', cr' equals $(v, cw)^{[0]}$.

```
$x, cr := v, cw$;
if $cr = cr'$
    then if $x_1 = x_2$
             then if $x_1 = x'_1 \neq x'_2 \wedge rtn = x'_2$
                      then skip
                      else $x' := x$;
                           $rtn := x_1$
                  fi
             else if $(x = x' \wedge rtn = x_2) \vee x'_1 = x'_2 = x_2$
                      then $x' := x$;
                           $rtn := x_2$
                      else $x' := x$;
                           $rtn := x_1$
                  fi
         fi
    else $x', cr' := x, cr$;
         $rtn := x_1$
fi
```

Figure 1.1: Construction 5: the reader's algorithm.

1.3 The Formal Model

1.3.1 System Executions

Almost all models of concurrent processes are based upon indivisible atomic actions as their primitive elements. For example, models in which a process is represented by a sequence or "trace" [1,12,13] assume that each element in the sequence represents an indivisible action. Net models [2] and related formalisms [9,10] assume that the firing of an individual transition is atomic. Operations to a nonatomic shared register cannot be modeled as atomic actions, since these formalisms have no concept of two atomic actions overlapping in time.

One can model a single read or write operation with two atomic actions: a *start* and a *finish* action. I will employ such a model to motivate the formalism. However, in the general view of physical systems based upon special relativity that is discussed in [7] and [5], there may be no single real event that precedes all other events in the operation, and no single event that follows all others. I will show that assuming such fictitious *start* and *finish* events would result in no loss of generality. However, it turns out to be easier to reason directly in terms of the nonatomic actions than to use starting and finishing events.

I therefore eschew more conventional formalisms in favor of one introduced in [6] and refined in [5], in which the primitive elements are *operation executions* that are not assumed to be atomic. In this formalism, an execution of a system is represented as a triple $S, \longrightarrow, \dashrightarrow$, where S is a finite or countably infinite set of operation executions, and $\longrightarrow$ and $\dashrightarrow$ are precedence relations on S.

The most general way of viewing the formalism is to consider an operation execution to be a set of points in four-dimensional space-time. Such a view is provided in [5]. While using the same formalism as [5], I will employ a less general but more intuitive model. In this model, an operation execution A is thought of as an activity performed during some time interval $[s_A, f_A]$, where the real numbers s_A and f_A are the starting and finishing times of A. I assume that at any time, only a finite number of operation executions have begun. Stated formally, a model consists of a set S of operation executions, together with real-valued functions s and f on S such that the following conditions hold for all A and B in S (where I write s_A and f_A instead of $s(A)$ and $f(A)$):

M1. $s_A \leq f_A$

M2. for any real number t: $\{A : s_A < t\}$ is finite

An operation execution A is said to be *instantaneous* if, for any $B \neq A$, the numbers s_B and f_B lie outside the interval $[s_A, f_A]$. Thus, A is instantaneous if and only if we can set s_A equal to f_A (shrinking the interval to a point) without changing the relative order of any starting and finishing times.

Given such a model, we can define the relations $\longrightarrow$ and $\dashrightarrow$ as follows:

$$\begin{aligned} A \longrightarrow B &\equiv f_A < s_B \\ A \dashrightarrow B &\equiv s_A \leq f_B \end{aligned} \tag{1}$$

Thus, $A \longrightarrow B$ means that A finishes before B starts, and $A \dashrightarrow B$ means that A starts no later than B finishes. We read $A \longrightarrow B$ as "A precedes B" and $A \dashrightarrow B$ as "A can affect B".

M1, M2 and (1) imply that the following hold for all operation executions A, B, C, and D in S:

A1. The relation $\longrightarrow$ is an irreflexive partial ordering.

A2. If $A \longrightarrow B$ then $A \dashrightarrow B$ and $B \not\dashrightarrow A$.

A3. If $A \longrightarrow B \dashrightarrow C$ or $A \dashrightarrow B \longrightarrow C$ then $A \dashrightarrow C$.

A4. If $A \longrightarrow B \dashrightarrow C \longrightarrow D$ then $A \longrightarrow D$.

A5. For any A, the set of all B such that $A \not\longrightarrow B$ is finite.

Instead of basing the formalism on this model, I adopt the more general view of [5] and take A1–A5 as axioms.

Definition 1 *A* system execution *is a triple $S, \longrightarrow, \dashrightarrow$ such that S is a finite or countably infinite set and $\longrightarrow$ and $\dashrightarrow$ are relations on S that satisfy A1–A5.*

Observe that A1 and A4 imply that if $A \longrightarrow B$ and $A \dashrightarrow B$ then $B \not\dashrightarrow A$, so the "and $B \not\dashrightarrow A$" in A2 is superfluous.

Definition 1 differs from the definition of a system execution given in [5] because I am considering only terminating operations. In the more general formalism, Axiom A5 needs the hypothesis that A terminates.

Definition 2 *A* global-time model *of a system execution* $S, \longrightarrow, \dashrightarrow$ *consists of a pair* s, f *of real-valued functions on* S *satisfying M1, M2 and (1). It is said to be* nondegenerate *if, for all* A*:* $s_A < f_A$ *and for all* $B \neq A$*:* $s_A \neq s_B$ *and* $s_A \neq f_B$.

A nondegenerate global-time model is one in which no two starting or stopping times are identical. The following result states that any global-time model can be turned into a nondegenerate one by tiny perturbations of the starting and finishing times of operation executions. Such perturbations should be allowed, since no physically meaningful result could depend upon completely accurate knowledge of these times. (It makes no physical sense to specify the starting and finishing times of an operation execution down to the fraction of a micropicosecond.)

Proposition 1 *For any any global-time model* s, f *of a system execution* $S, \longrightarrow, \dashrightarrow$ *and any* $\epsilon > 0$*, there exists a nondegenerate global-time model* s', f' *of* $S, \longrightarrow, \dashrightarrow$ *such that* $|s'_A - s_A| < \epsilon$ *and* $|f'_A - f_A| < \epsilon$ *for all* $A \in S$.

The proofs of this and all other propositions stated in this section are given in the appendix.

In a global-time model, the starting and finishing times of operations are totally ordered. Given two operation executions A and B, s_B must be either greater than or not greater than f_A, so the following condition holds.

A#. For any operation executions A and B with $A \neq B$: $A \longrightarrow B$ or $B \dashrightarrow A$.

This condition does not hold for all system executions. (Trivial counterexamples are obtained by noting that the empty precedence relations make any set a system execution.) Condition A# holds only if there is global-time model.

Proposition 2 *A system execution $S, \longrightarrow, \dashrightarrow$ has a global-time model if and only if A# holds.*

In the more general interpretation of operation executions given in [5], condition A# fails to hold for a pair of operation executions A, B if A and B occur at spatially separated locations, and they both happen within a time interval that is less than the time needed for light to travel between their locations. In most systems of practical interest, A# holds for almost all pairs A, B of operation executions.

The following result shows that we can get a global-time model by adding extra precedence relations.

Proposition 3 *Given any system execution $S, \longrightarrow, \dashrightarrow$, there exist extensions $\stackrel{\prime}{\longrightarrow}$ of $\longrightarrow$ and $\stackrel{\prime}{\dashrightarrow}$ of $\dashrightarrow$ such that $S, \stackrel{\prime}{\longrightarrow}, \stackrel{\prime}{\dashrightarrow}$ is a system execution satisfying A#.*

Later, I will indicate why we can consider the system execution $S, \longrightarrow, \stackrel{\prime}{\dashrightarrow}$ to be a reasonable way of viewing the system execution $S, \longrightarrow, \dashrightarrow$.

A system execution satisfying A# is maximal in the sense that no additional $\longrightarrow$ or $\dashrightarrow$ relations can be added. This is because, for any pair of distinct operation executions A and B, A# implies that either $A \longrightarrow B$, or $B \longrightarrow A$, or $A \dashrightarrow B$ and $B \dashrightarrow A$. In any of these three cases, adding an additional precedence relation would violate A1 or A2.

When trying to understand an algorithm or its correctness proof, it is useful to think in terms of a global-time model, drawing pictures of reads and writes as time intervals. However, I find that the best way to formalize the proof is to use Axioms A1–A5. The additional assumption A#, implicitly introduced when using a global-time model, is not needed.

1.3.2 Hierarchial Views

The same system can be viewed at different levels of detail, with different operation executions at each level. Viewed at the customer's level, a banking system has operation executions such as ***deposit*** \$10. Viewed at the programmer's level, this same system executes operations such as *dep_amt*[*cust*] := 1000. The fundamental problem of system building is to implement one system (like a banking system) as a higher-level view of another system (like a Pascal program).

A higher-level operation consists of a set of lower-level operations—the set of operations that implement it. Let $S, \longrightarrow, \dashrightarrow$ be a system execution and let $\mathcal{H}$ be a set whose elements, called ***higher-level operation executions***, are sets of operation executions from S. We consider the starting time $s^{\bullet}_{H}$ of a higher-level operation execution H to be the earliest starting time of all the operation executions it contains, and its finishing time $f^{\bullet}_{H}$ to be their latest finishing time. In other words, for every H in $\mathcal{H}$:

$$\begin{aligned} s^{\bullet}_{H} &= \min\{s_A : A \in \mathcal{H}\} \\ f^{\bullet}_{H} &= \max\{f_A : A \in \mathcal{H}\} \end{aligned} \tag{2}$$

In order for this to define real-valued functions $s^{\bullet}$ and $f^{\bullet}$ on $\mathcal{H}$ that satisfy M1 and M2, it is sufficient for $\mathcal{H}$ to satisfy the following two conditions:

H1. Each element of $\mathcal{H}$ is a finite, nonempty set of elements of S.

H2. Each element of S belongs to a finite, nonzero number of elements of $\mathcal{H}$.

A set $\mathcal{H}$ of subsets of S satisfying H1 and H2 is called a ***higher-level view*** of S. In most cases of interest, $\mathcal{H}$ is a partition of S, so each element of S belongs to exactly one element of $\mathcal{H}$. However, I allow the more general case in which a single lower-level operation execution is viewed as part of the implementation of more than one higher-level one.

Let $S, \longrightarrow, \dashrightarrow$ be a system execution with a global-time model s, f, and let $\mathcal{H}$ be a higher-level view of S. We can define $s^{\bullet}$ and $f^{\bullet}$ by (2) and then use (1) to define $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$, obtaining a system execution $\mathcal{H}, \overset{\bullet}{\longrightarrow}, \overset{\bullet}{\dashrightarrow}$ having $s^{\bullet}, f^{\bullet}$ as a global-time model. The precedence relations $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$ can be obtained directly from $\longrightarrow$ and $\dashrightarrow$ as follows:

$$\begin{aligned} G \overset{\bullet}{\longrightarrow} H &\equiv \forall A \in G : \forall B \in H : A \longrightarrow B \\ G \overset{\bullet}{\dashrightarrow} H &\equiv \exists A \in G : \exists B \in H : A \dashrightarrow B \text{ or } A = B \end{aligned} \tag{3}$$

We can forget about the global-time models and take (3) to be the definitions of $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$. It is easy to show that if $\mathcal{H}$ satisfies H1 and H2, and

$\longrightarrow$ and $\dashrightarrow$ satisfy A1–A5, then $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$ also satisfy A1–A5. Therefore, if $\mathcal{H}$ is a higher-level view of $\mathcal{S}$, then $\mathcal{H}, \overset{\bullet}{\longrightarrow}, \overset{\bullet}{\dashrightarrow}$ is a system execution. If the relations $\longrightarrow$ and $\dashrightarrow$ also satisfy A#, then so do $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$.

Let us now consider what it means for one system to implement another. If the system execution $\mathcal{S}, \longrightarrow, \dashrightarrow$ is an implementation of a system execution $\mathcal{S}, \overset{\mathcal{H}}{\longrightarrow}, \overset{\mathcal{H}}{\dashrightarrow}$, then we expect $\mathcal{H}$ to be a higher-level view of $\mathcal{S}$—that is, each operation in $\mathcal{H}$ should consist of a set of operation executions of $\mathcal{S}$ satisfying H1 and H2. This describes the elements of $\mathcal{H}$, but not the precedence relations $\overset{\mathcal{H}}{\longrightarrow}$ and $\overset{\mathcal{H}}{\dashrightarrow}$. What should those relations be?

If we consider the system execution $\mathcal{S}$ to be the "real" one and $\mathcal{H}$ to be a fictitious grouping of the real operation executions into abstract, higher-level ones, then the induced relations $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$ are the "real" precedence relations on $\mathcal{H}$. These induced relations make the higher-level view $\mathcal{H}$ a system execution, so they are an obvious choice for the relations $\overset{\mathcal{H}}{\longrightarrow}$ and $\overset{\mathcal{H}}{\dashrightarrow}$. However, they may not be the proper choice. Suppose that we are trying to implement an atomic register using several simpler ones, and consider a read R and write W to that register—that is, R and W are operation executions in $\mathcal{H}$ that represent a read and write to the register. Atomicity means that either $R \overset{\mathcal{H}}{\longrightarrow} W$ or $W \overset{\mathcal{H}}{\longrightarrow} R$. However, the two operation executions could really be concurrent. For example, there could be some operation executions A and B in the implementation of R and an operation execution C in the implementation of W with $A \longrightarrow C \longrightarrow B$, which (by (3)) implies $R \overset{\bullet}{\dashrightarrow} W$ and $W \overset{\bullet}{\dashrightarrow} R$. Thus, (by A2) the induced relations $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$ cannot be the desired relations $\overset{\mathcal{H}}{\longrightarrow}$ and $\overset{\mathcal{H}}{\dashrightarrow}$.

When implementing an atomic register from nonatomic ones, in addition to specifying what set of lower-level operation executions corresponds to an atomic read or write, one must also specify how to determine whether a read, which may really be concurrent with a write (according to the induced relations $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$), is considered to precede or follow that write. This must be specified in such a way that the register satisfies the condition of atomicity—namely, that each read obtains the value written by the most recent write. Subject to that requirement, there is a great deal of freedom in specifying the high-level relation $\overset{\mathcal{H}}{\longrightarrow}$.

The implementor cannot be completely free to specify the precedence relations in the high-level system any way he wishes. For example, if there is at least one write of every possible value of the register, then any system execution can be viewed as the implementation of an atomic register by choosing the $\xrightarrow{\mathcal{H}}$ relation to be a sequential ordering of the reads and writes in which every read comes between any write of the value it read and the next write operation. This could lead to a precedence relation in which an operation is defined to precede one that really occurred several months earlier. Such a precedence relation obviously seems absurd, but why? In a real system, these reads and writes occur deep within the computer; we never actually see them happen. What is wrong with defining the precedence relation $\xrightarrow{\mathcal{H}}$ to pretend that these operation executions happened in any order we wish? After all, we are already pretending, contrary to fact, that the operations are not concurrent.

In addition to reads and writes to registers, real systems perform externally observable operation executions such as printing on terminals. By observing these operation executions, we can infer some precedence relations among the internal reads and writes. We need some condition on $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ to rule out precedence relations that contradict such observations.

These contradictions are avoided by requiring that the interval in which we pretend an operation execution occurs (in forming the $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ relations) be contained within the interval in which it actually occured. In other words, we require that a global-time model $s^{\mathcal{H}}, f^{\mathcal{H}}$ for $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ satisfy

$$s^{\bullet}_A \leq s^{\mathcal{H}}_A \leq f^{\mathcal{H}}_A \leq f^{\bullet}_A \qquad (4)$$

where $s^{\bullet}$ and $f^{\bullet}$ are defined by (2). To reformulate (4) directly in terms of the precedence relations, I appeal to the following result.

Proposition 4 *Let s, f be a nondegenerate global-time model for a system execution $S, \longrightarrow, \dashrightarrow$ and let $S, \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ be a system execution satisfying A# such that for any $A, B \in S$: $A \longrightarrow B$ implies $A \xrightarrow{\prime} B$. Then there exists a nondegenerate global-time model s', f' for $S, \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ such that for all $A \in S$:*

$$s_A \leq s'_A < f'_A \leq f_A$$

This result implies that, if the system executions $S, \longrightarrow, \dashrightarrow$ and $\mathcal{H}, \xrightarrow{\mathcal{H}},$

$\overset{\mathcal{H}}{\dashrightarrow}$ both satisfy A#, then the ability to choose $s^{\mathcal{H}}$ and $f^{\mathcal{H}}$ satisfying (4) is equivalent to the following condition:

H3. For any $G, H \in \mathcal{H}$: if $G \xrightarrow{\bullet} H$ then $G \xrightarrow{\mathcal{H}} H$, where $\xrightarrow{\bullet}$ is defined by (3).

This should serve to motivate the following formal definition, which does not mention global-time models.

Definition 3 *A system execution* $\mathcal{S}, \longrightarrow, \dashrightarrow$ implements *a system execution* $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ *if H1–H3 are satisfied.*

To relate this definition to the preceding discussion of observable operation executions, we need the following result. Its statement relies upon the obvious fact that if $\mathcal{S}, \longrightarrow, \dashrightarrow$ is a system execution, then $\mathcal{T}, \longrightarrow, \dashrightarrow$ is also a system execution for any subset $\mathcal{T}$ of $\mathcal{S}$. (The symbols $\longrightarrow$ and $\dashrightarrow$ denote both the relations on $\mathcal{S}$ and their restrictions to $\mathcal{T}$. Also, in the proposition, the set $\mathcal{T}$ is identified with the set of all singleton sets $\{A\}$ for $A \in \mathcal{T}$.)

Proposition 5 *Let* $\mathcal{S} \cup \mathcal{T}, \longrightarrow, \dashrightarrow$ *be a system execution, where* $\mathcal{S}$ *and* $\mathcal{T}$ *are disjoint; let* $\mathcal{S}, \longrightarrow, \dashrightarrow$ *be an implementation of a system execution* $\mathcal{H}$, $\xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$; *and let* $\xrightarrow{\bullet}$ *and* $\overset{\bullet}{\dashrightarrow}$ *be the relations defined on* $\mathcal{H} \cup \mathcal{T}$ *by (3). Then there exist precedence relations* $\xrightarrow{\mathcal{H}\mathcal{T}}$ *and* $\overset{\mathcal{H}\mathcal{T}}{\dashrightarrow}$ *such that:*

- $\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{H}\mathcal{T}}, \overset{\mathcal{H}\mathcal{T}}{\dashrightarrow}$ *is a system execution that is implemented by* $\mathcal{S} \cup \mathcal{T}, \longrightarrow, \dashrightarrow$.
- *The restrictions of* $\xrightarrow{\mathcal{H}\mathcal{T}}$ *and* $\overset{\mathcal{H}\mathcal{T}}{\dashrightarrow}$ *to* $\mathcal{H}$ *equal* $\xrightarrow{H}$ *and* $\overset{H}{\dashrightarrow}$, *respectively.*
- *The restrictions of* $\xrightarrow{\mathcal{H}\mathcal{T}}$ *and* $\overset{\mathcal{H}\mathcal{T}}{\dashrightarrow}$ *to* $\mathcal{T}$ *are extensions of the relations* $\xrightarrow{\bullet}$ *and* $\overset{\bullet}{\dashrightarrow}$, *respectively.*

To apply this proposition to our discussion of implementations, let $\mathcal{S}, \longrightarrow, \dashrightarrow$ be an execution of a lower-level system of register reads and writes implementing a higher-level system execution $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ of reads and writes. Let $\mathcal{T}$ be the set of all other operation executions in the system, including the observable ones. Proposition 5 means that, while the precedence relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ may imply new precedence relations on the operation executions in $\mathcal{T}$, these relations ($\xrightarrow{\mathcal{H}\mathcal{T}}$ and $\overset{\mathcal{H}\mathcal{T}}{\dashrightarrow}$) are consistent with the "real" precedence relations $\xrightarrow{*}$ and $\overset{*}{\dashrightarrow}$ on T.

Note that when there are global-time models for all the system executions, the $*$ relations are the same as the original precedence relations on the set $\mathcal{T}$, and Proposition 4 implies that the $\mathcal{H}\mathcal{T}$ relations can be chosen also to be the same as the original precedence relations on $\mathcal{T}$. However, in general, the relation $\xrightarrow{\mathcal{H}}$ may contain orderings that imply additional orderings on the elements of $\mathcal{T}$ beyond those contained in $\xrightarrow{*}$. As a simple example, let $A, B \in \mathcal{S}$, let $S, T \in \mathcal{T}$, let $S \longrightarrow A$, $B \longrightarrow T$ be the only precedence relations among these elements, and let $\mathcal{H} = \mathcal{S}$. If $A \xrightarrow{\mathcal{H}} B$, then A1 implies $S \xrightarrow{\mathcal{H}\mathcal{T}} T$ even though $S \overset{*}{\not\longrightarrow} T$.

When implementing a register, I will ignore any operation executions not involved in the implementation, and consider the system execution comprised only of the reads and writes that implement the register. Proposition 5 shows that the implementation cannot lead to any anomalous precedence relations among the operation executions that are being ignored.

An implementation $\mathcal{S}, \longrightarrow, \dashrightarrow$ of $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ is said to be *trivial* if every element of $\mathcal{H}$ is a singleton set. In other words, a trivial implementation is one in which each higher-level operation execution is implemented by a single lower-level one. In a trivial implementation, the sets $\mathcal{S}$ and $\mathcal{H}$ are (essentially) the same; the two system executions differ only in their precedence relations.

Proposition 3 implies that any system execution trivially implements one that satisfies A#, which, by Proposition 2, has a global-time model. Implementation is transitive—if $\mathcal{S}, \longrightarrow, \dashrightarrow$ implements $\mathcal{S}', \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ which in turn implements $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$, then $\mathcal{S}, \longrightarrow, \dashrightarrow$ implements $\mathcal{H}, \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$. When implementing a higher-level system, we can therefore assume the lower-level system execution has a global-time model. However, there is no reason to do so; a rigorous correctness proof using Axioms A1–A5 will be at least as simple as one based upon starting and finishing times, and will be more reliable than an intuitive one based upon pictures of intervals.

1.3.3 Register Axioms

The foregoing discussion applies to any system execution. I now consider system executions containing reads and writes to registers. In addition to A1–A5, some axioms special to these kinds of operation executions are needed, including axioms that provide the formal definitions of safe, regular, and atomic registers.

Axioms A1–A5 do not require that there be any precedence relations among operation executions. However, some precedence relation between a read and a write to the same register must be assumed. (Communication requires a causal connection between reads and writes.) The following axiom is assumed; the reader is referred to [5] (where it is labeled C3) for its justification. Note that it is implied by A#.

B1. For any read R and write W to the same register, $R \dashrightarrow W$ or $W \dashrightarrow R$ (or both).

Each register is assumed to have a finite set of possible values—for example, a boolean-valued register has the possible values *true* and *false*. I assume that any read, whether or not it overlaps a write, obtains one of these values.

B2. A read of a register obtains one of the values that may be written in the register.

Thus, a read of a Boolean register cannot obtain a nonsense value like "*trlse*". This axiom does not assume that the value obtained by a read was ever actually written in the register.

I assume that a register v is written by only a single writer, and that each write precedes the next. Let $V^{[1]}, V^{[2]}, \ldots$ denote the sequence of write operations to the register v, where

$$V^{[1]} \longrightarrow V^{[2]} \longrightarrow \cdots$$

and let $v^{[i]}$ denote the value written by $V^{[i]}$. (There may be a finite or infinite number of write operations $V^{[i]}$.)

A register v is assumed to have some initial value $v^{[0]}$. It is convenient to assume that this value is written by a write $V^{[0]}$ that precedes ($\longrightarrow$) all other reads and writes of v. Eliminating this assumption changes none of the results, but it complicates the reasoning because a read that precedes all writes has to be treated as a separate case.

Let R be a read of register v, and let

$$\begin{aligned} I_R &\overset{\text{def}}{=} \{V^{[k]} : R \not\dashrightarrow V^{[k]}\} \\ J_R &\overset{\text{def}}{=} \{V^{[k]} : V^{[k]} \dashrightarrow R\} \end{aligned}$$

It follows from A2 and the assumption that $V^{[0]}$ precedes all reads that $V^{[0]}$ is in both I_R and J_R; and it follows from A2 and A5 that I_R and J_R are finite. The writes in J_R are the ones that could affect R. For the sake of the following intuitive discussion, suppose that A# holds, so I_R is the set of writes that precede ($\longrightarrow$) R. (The reader interested in extending his intuition to the general case should substitute "precedes" by "effectively precedes"—a concept defined in [5].) The difference $J_R - I_R$ of these two sets is the set of writes concurrent with R. The read R can observe "traces" of the values written by writes in $J_R - I_R$, and by the last write in I_R. All traces of earlier writes are assumed to vanish with the completion of the last write in I_R, and no write later than the last one in J_R can influence R in any way.

I will say that R *sees* $v^{[i,j]}$ if it can observe traces of the writes $V^{[i]}$ through $V^{[j]}$. The formal definition is as follows:

Definition 4 *A read R of register v is said to* see $v^{[i,j]}$ *where:*

$$\begin{aligned} i &\overset{\text{def}}{=} \max\{k : R \not\dashrightarrow V^{[k]}\} \\ j &\overset{\text{def}}{=} \max\{k : V^{[k]} \dashrightarrow R\} \end{aligned}$$

This definition makes sense because i and j are defined to be the maxima of finite, nonempty sets—A5 and A2 imply that they are finite, and they both contain zero. Also observe that B1 implies that $i \leq j$.

I can now give the formal definitions of safe, regular, and live registers. A safe register is one that obtains the correct value if it is not concurrent with any write. This is the case if it observes traces of only a single write.

B3. (*safe*) A read that sees $v^{[i,i]}$ obtains the value $v^{[i]}$.

A regular register is one that obtains a value that it "could have" seen.

B4. (*regular*) A read that sees $v^{[i,j]}$ obtains a value $v^{[k]}$ for some k with $i \leq k \leq j$

An atomic register satisfies the additional requirement that a read is never concurrent with any write.

B5. (*atomic*) If a read sees $v^{[i,j]}$ then $i = j$.

A safe register satisfies B1–B3, a regular register satisfies B1–B4 (note that B4 implies B3), and an atomic register satisfies B1–B5.

The following two propositions state some useful properties that are simple consequences of Definition 4. I introduce the notation of letting $v^{[i,j]}$ stand for a read that sees the value $v^{[i,j]}$. Thus, part (a) is an abbreviation for: "If R is a read that sees $v^{[i,j]}$ and $R \longrightarrow V^{[k]}$ then ...." (Recall that $V^{[k]}$ is the k^{th} write of v.)

Proposition 6 (a) *If* $v^{[i,j]} \longrightarrow V^{[k]}$ *then* $j < k$.

(b) *If* $V^{[k]} \longrightarrow v^{[i,j]}$ *then* $k \leq i$.

(c) *If* $v^{[i,j]} \longrightarrow v^{[i',j']}$ *then* $j \leq i' + 1$.

Proposition 7 *If R is a read that sees* $v^{[i,j]}$, *then*

(a) $k \leq j$ *if and only if* $V^{[k]} \dashrightarrow R$.

(b) $i \leq k$ *if and only if* $R \dashrightarrow V^{[k+1]}$.

In a global-time view, atomicity is usually defined to mean that all operations are instantaneous. In B5, it is defined by the requirement that a write does not overlap a read. However, two reads may overlap, and a write could overlap some operation execution that is not a read or write of the register. It is easy to see that, given a global-time model for a system execution satisfying B5, without violating conditions B1–B5, we can shrink the intervals occupied by reads and writes so that they overlap no other operations. Thus, the original system execution implements one in which reads and writes of the atomic register are instantaneous.

For a nonatomic register, reads and writes cannot be made instantaneous. However, the reads can be made instantaneous.

Proposition 8 *Any system execution $S, \longrightarrow, \dashrightarrow$ having a safe or regular register v trivially implements a system execution $S, \overset{\prime}{\longrightarrow}, \overset{\prime}{\dashrightarrow}$ in which v is also safe or regular, such that $S, \overset{\prime}{\longrightarrow}, \overset{\prime}{\dashrightarrow}$ has a global-time model in which every read of v is instantaneous.*

I have observed that a regular register is not necessarily atomic because two successive reads that overlap the same write could return the new then the old value. The following result shows that this is the only way a regular register can fail to be atomic.

Proposition 9 *Let $S, \longrightarrow, \dashrightarrow$ be a system execution containing reads and writes to a regular register v, and let ϕ be an integer-valued function on the set of reads such that:*

1. *If R sees $v^{[i,j]}$, then $i \leq \phi(R) \leq j$.*
2. *A read R returns the value $v^{[\phi(R)]}$.*
3. *If $R \longrightarrow R'$ then $\phi(R) \leq \phi(R')$.*

Then $S, \longrightarrow, \dashrightarrow$ trivially implements a system execution in which v is an atomic register.

A function ϕ satisfying the first two properties exists if and only if v is regular. One might be tempted to replace these three properties with the requirement that v be regular and the following hold:

3′ If $v^{[i,j]} \longrightarrow v^{[i',j']}$ then there exist k and k' with $i \leq k \leq j$ and $i' \leq k' \leq j'$ such that $v^{[i,j]}$ returns the value $v^{[k]}$ and $v^{[i',j']}$ returns the value $v^{[k']}$.

However, this does not imply atomicity. As a counterexample, let $v^{[0]} = v^{[2]} = 0$ and $v^{[1]} = 1$, let R_1, R_2, R_3 be the three reads shown in Figure 1.2, and suppose that R_1 and R_3 return the value 1 while R_2 returns the value

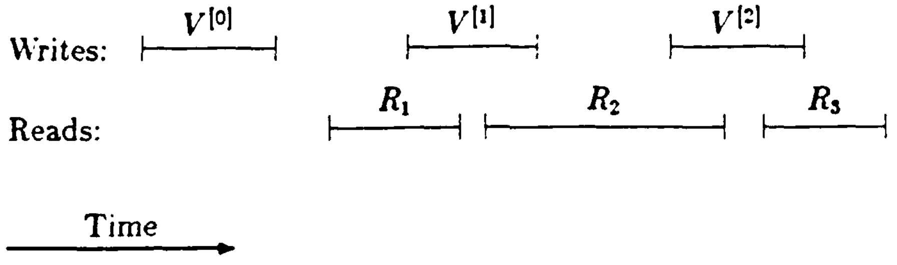


Figure 1.2: An interesting collection of reads and writes.

0. The reader can show that this register is regular, but no such ϕ can be constructed; there is no way to interpret these reads and writes as belonging to an atomic register while maintaining the given orderings among the writes and among the reads.

If two reads cannot overlap the same write, then $v^{[i,j]} \longrightarrow v^{[i',j']}$ implies $j \leq i'$. This implies that any ϕ satisfying conditions 1 and 2 of Proposition 9 also satisfies condition 3. But such a ϕ exists if v is regular, so any regular register trivially implements an atomic one if two reads cannot overlap a single write.

1.3.4 Systems

I have defined a system execution, but not a system. Formally, a system is just a set of system executions—a set that represents all possible executions of the system.

Definition 5 *A* system *is a set of system executions. The system* **S** *is said to contain a register v satisfying one or more of the properties B1–B5 if every system execution in* **S** *contains a sequence* $V^{[1]} \longrightarrow \cdots$ *of writes with associated values* $v^{[1]}, \ldots$ *and a set of reads satisfying the corresponding properties.*

The usual method of describing a system is with a program written in some programming language. Each execution of such a program describes a system execution, and the program represents the system consisting of the set of all such executions. The only operation executions that concern us are reads and writes of a register; "calculation" steps can be ignored.

For example, execution of the statement $x := y \vee z$ includes three operation executions: a read of y, a read of z, and a write of x. It does not matter whether or not the computation of the $\vee$ is considered to be a separate operation execution. What is significant is that each of the two reads precedes ($\longrightarrow$) the write; no precedence relation is assumed between the two reads.

A formal semantics for a programming language can be given by defining, for each syntatically correct program, the set of all possible executions. This is done by recursively defining a succession of lower and lower higher-level views, in which each operation execution represents a single execution of a syntactic program unit.[2] At the highest-level view, a system execution consists of a single operation execution that represents an execution of the entire program. A view in which an execution of the statement $S; T$ is a single operation execution is refined into one in which an execution consists of an execution of S followed by ($\longrightarrow$) an execution of T.[3] While this kind of formal semantics may be useful in studying subtle programming language issues, it is unnecessary for the simple language constructs used in the algorithms of this paper, so I will just employ these ideas informally.

Having defined what a system is, I should define what it means for one system to implement another. The definition is, of course, in terms of the definition of what it means for one system execution to implement another.

Definition 6 *The system* **S** implements *a system* **H** *if there is a mapping* $\iota : \mathbf{S} \mapsto \mathbf{H}$ *such that, for every system execution* $\langle S, \longrightarrow, \dashrightarrow \rangle$ *in* **S**, $\langle S, \longrightarrow, \dashrightarrow \rangle$ *implements* $\iota(\langle S, \longrightarrow, \dashrightarrow \rangle)$.

Note that for **S** to implement **H**, every execution of **S** must correspond to some execution of **H**. The converse is not required; I do not insist that every possible execution of **H** have a corresponding implementation. A higher-level description **H** of a system can be viewed as a specification of

[2] For nonterminating programs, the formalism must be extended to allow a nonterminating higher-level operation execution that consist of an infinite set of lower-level operation executions.

[3] In the general case, we must also allow the possibility that an execution of $S; T$ consists of a nonterminating execution of S.

its implementation—a specification that describes all allowed behaviors, but does not require any particular behavior.

This definition raises the question of how we can specify that the system must actually do anything. The specification of a banking system must allow a possible system execution in which no customers happen to use an automatic teller machine on a particular afternoon, and it must include the possibility that a customer will enter an invalid request. How can we rule out an implementation in which the machine simply ignores all customer requests during an afternoon, or interprets any request as an invalid one?

The answer lies in the concept of an *interface specification*, discussed in [8]. The specification must explicitly describe how certain interface operations are to be implemented; their implementation is not left to the implementor. The interface specification for the bank includes a description of what sequences of keystrokes at the teller machine constitute valid requests, and the set of system executions only includes ones in which every valid request is serviced. What it means for someone to use the machine is part of the interface specification, so the possibility of no one using the machine on some afternoon does not allow the implementation to ignore someone who does use it.

Since this paper considers only the internal operations that effect communication between processes within the system, not the interface operations that effect communication between the system and its environment, I will ignore interface specifications. The interested reader is referred to [8] for a discussion of this subject.

1.4 Correctness Proofs for the Constructions

1.4.1 Proof of Constructions 1, 2 and 3

These constructions are all simple, and the correctness proofs are essentially trivial. Formal proofs add no further insight into the constructions, but they do illustrate how the formalism developed in the preceding section is applied to actual algorithms. I therefore indicate all the formal details in the proof of Construction 1. The formal proofs for the other two constructions are just briefly sketched.

Recall that in Construction 1, the m-reader register v is implemented by the m single-reader registers v_i. Formally, this construction defines a system, which I denote by **S**, that is the set of all system executions consisting of reads and writes of the v_i such that the only operations to these registers are the ones indicated by the readers' and writer's programs. Thus, **S** consists of all system executions $S, \longrightarrow, \dashrightarrow$ such that:

- S consists of reads and writes of the registers v_i.
- Each v_i is written by the same writer and is read only by the i^{th} reader.
- For any i and j: if the write $V_i^{[k]}$ occurs then the write $V_j^{[k]}$ also ocurs, and $v_i^{[k-1]} \longrightarrow v_j^{[k]}$.

The third condition expresses the formal semantics of the writer's algorithm, asserting that a write of v is done by writing all the v_i, and that a write of v is completed before the next one is begun.

To say that the v_i are safe or regular means that the system **S** is further restricted to contain only system executions that satisfy B1–B3 or B1–B4, when each v_i is substituted for v in those conditions.

To show that this construction implements a register v, Definition 6 states that we must construct a mapping ι from **S** to the system **H**, which consists of the set of all system executions formed by reads and writes to an m-reader register v. To say that v is safe or regular means that **H** contains only system executions satisfying B1–B3 or B1–B4.

In giving the readers' and writer's algorithms, the construction implies that for each system execution $S, \longrightarrow, \dashrightarrow$ of S, the set $\iota(S)$ of operation executions of $\iota(S, \longrightarrow, \dashrightarrow)$ is the higher-level view of $S, \longrightarrow, \dashrightarrow$ consisting of all writes $V^{[k]}$ of the form $\{V_1^{[k]}, \ldots, V_m^{[k]}\}$, for $V_i^{[k]} \in S$, and all reads of the form $\{R_i\}$, where $R_i \in S$ is a read of v_i. (The write $V^{[k]}$ exists in $\iota(S)$ if and only if some, and hence all, $V_i^{[k]}$ exists.) Conditions H1 and H2 are obviously satisfied, so this is indeed a higher-level view. To complete the mapping ι, we must define the precedence relations $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ so that $\iota(S, \longrightarrow, \dashrightarrow)$ is defined to be $\iota(S), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$. Proving the correctness of the construction means showing that:

1. $\iota(S), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ is a system execution—that is, it satisfies A1–A5.

2. $S, \longrightarrow, \dashrightarrow$ implements $\iota(S), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$—that is, H1–H3 are satisfied.

3. $\iota(S), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$ is in **H**—that is, B1–B3 or B1–B4 are satisfied.

The precedence relations on $\iota(S)$ are defined to be the "real" ones, with $G \xrightarrow{\mathcal{H}} H$ if and only if G really precedes H. Formally, this means that we let $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$ be the induced relations $\xrightarrow{\bullet}$ and $\overset{\bullet}{\dashrightarrow}$, defined by (3). Recall from Section 3.2 that the induced precedence relations make any higher-level view a system execution, so 1 is satisfied. I have already observed that H1 and H2, which are independent of the choice of precedence relations, are satisfied, and H3 is trivially satisfied by the induced precedence relations, so 2 holds. Therefore, we need only show that if B1–B3 or B1–B4 are satisfied for reads and writes of each of the registers v_i in $S, \longrightarrow, \dashrightarrow$, then they are also satisfied by the register v of $\iota(S), \xrightarrow{\mathcal{H}}, \overset{\mathcal{H}}{\dashrightarrow}$.

Property B1 for $\iota(S), \xrightarrow{\bullet}, \overset{\bullet}{\dashrightarrow}$ follows easily from (3) and property B1 for $S, \longrightarrow, \dashrightarrow$. Property B2 is immediate. The informal proof of B3 is as follows: if a read of v by process i does not overlap a write (in $\iota(S)$), then the read of v_i does not overlap any write of v_i, so it obtains the correct value. A formal proof is based upon:

X. If a read R_i in $S, \longrightarrow, \dashrightarrow$ sees $v_i^{[k,l]}$, then the corresponding read $\{R_i\}$ in $\iota(S), \xrightarrow{\bullet}, \overset{\bullet}{\dashrightarrow}$ sees $v^{[k',l']}$, where $k' \leq k \leq l \leq l'$.

The proof of X is a straightforward application of (3) and Defintion 4. Property X easily implies that if B3 or B4 holds for $S, \longrightarrow, \dashrightarrow$, then it holds for $\iota(S), \overset{\bullet}{\longrightarrow}, \overset{\bullet}{\dashrightarrow}$. This completes the formal proof of Construction 1.

The formal proof of Construction 2 is quite similar. Again, the induced precedence relations are used to turn a higher-level view into a system execution. The proof of Construction 3 is a bit trickier because a write operation to v^{*} that does not change its value consists only of the read operation to the internal variable x. This means that the induced precedence relations do not necessarily satisfy B1; they must be extended to make B1 hold. This can be done by applying Proposition 3, though a more "economical" extension can also be constructed.

1.4.2 Proof of Construction 4

The higher-level system execution of reads and writes to v is defined to have the induced precedence relations $\overset{\bullet}{\longrightarrow}$ and $\overset{\bullet}{\dashrightarrow}$. As in the above proofs, verifying that this defines an implementation and that B1 holds is trivial. The only problems are proving B2—namely, showing that the reader must find some v_i equal to one—and proving B4 (which implies B3).

I first prove the following property:

Y. If a read returns the value μ, then there is some k such that $v^{[k]} = \mu$ and the read sees $v^{[l,r]}$ with $l \leq k \leq r$.

If B2 holds, then property Y implies B4.

Reasoning about the construction is complicated by the fact that a write of v does not write all the v_j, so the write of v_j that occurs during the kth write of v is not necessarily the kth write of v_j. To overcome this difficulty, I introduce new names for the write operations to the v_j. If v_j is written during the execution of $V^{[k]}$, then I let $W_j^{[k]}$ denote that write of v_j; otherwise, $W_j^{[k]}$ is undefined. Thus, every write $V_j^{[l]}$ of v_j is also named $W_j^{[l']}$ for some $l' \geq l$. I will say that a read of v_j sees $w_j^{[l',r']}$ if it sees $v^{[l,r]}$ and the writes $W_j^{[l']}$ and $W_j^{[r']}$ are the same writes as $V_j[l]$ and $V_j[r]$, respectively. Note that, because the writer's algorithm writes from "right to left", if $W_i^{[k]}$ exists, then so do all the $W_j^{[k]}$ with $j < i$. In particular, $W_1^{[k]}$ exists for all k.

Let R be a read that returns the value μ, and let μ be the ith value, so R consists of the sequence of reads $R_1 \longrightarrow \cdots \longrightarrow R_i$, where each R_j is a read of v_j. All the R_j return the value 0 except R_i, which returns the value 1. Let R see $v^{[l,r]}$ and let each R_j see $w_j^{[l(j),r(j)]}$. By regularity of v_j, there is some $k(j)$ with $l(j) \leq k(j) \leq r(j)$ such that $W_i^{[k(i)]}$ writes a 1 and $W_j^{[k(j)]}$ writes a 0 for $1 \leq j < i$. Thus, $v^{[k(i)]}$ is the value read by R, so it suffices to show that $l \leq k(i) \leq r$.

Definition 4 implies $W_i^{[r(i)]} \dashrightarrow R_i$, which by (3) implies $V^{[r(i)]} \overset{\bullet}{\dashrightarrow} R$, which implies $r(i) \leq r$. Hence, $k(i) \leq r$.

For any p with $p \leq l$, Definition 4 implies that $R \overset{\bullet}{\not\dashrightarrow} V^{[p]}$, which implies that $R_1 \not\dashrightarrow W_1^{[p]}$, which in turn implies that $p \leq l(1)$. Hence, $l \leq l(1)$.[4] Since $l(j) \leq k(j)$, it suffices to prove that $k(j) \leq l(j+1)$ for $1 \leq j < i$.

Since $k(j) \leq r(j)$, Definition 4 implies that $W_j^{[k(j)]} \dashrightarrow R_j$. Because $W_j^{[k(j)]}$ writes a zero, $W_{j+1}^{[k(j)]}$ exists, and we have

$$W_{j+1}^{[k(j)]} \longrightarrow W_j^{[k(j)]} \dashrightarrow R_j \longrightarrow R_{j+1}$$

where the two $\longrightarrow$ relations are implied by the order in which writing and reading of the individual v_j are performed. By A4, this implies that $W_{j+1}^{[k(j)]} \longrightarrow R_{j+1}$, which, by A2, implies $R_{j+1} \not\dashrightarrow W_{j+1}^{[k(j)]}$. By Definition 4, this implies that $k(j) \leq l(j+1)$, completing the proof of property Y.

To complete the proof of the construction, I must only prove that every read does return a value. Let R and the values $l(j)$, $k(j)$, and $r(j)$ be as above, except let $i = n$ and drop the assumption that R_i obtains the value 1. To prove B2, I must prove that R_n does obtain the value 1.

The same argument used above shows that if R_j obtains a zero, then that zero was written by some write $W_j^{[k(j)]}$, which implies that $W_{j+1}^{[k(j)]}$ exists and $k(j) \leq l(j+1)$. Since R_n obtains the value written by $W_n^{[k(n)]}$, it must obtain a 1 unless $k(n) = 0$ and the initial value is not the nth one. Suppose the initial value $v^{[0]}$ is the pth value, encoded with $v_p = 1$, $p < n$. Since R_p obtains the value 0, we must have $k(p) > 0$, which implies that $k(n) > 0$, so R_n obtains the value 1. This completes the proof of the construction.

[4]Note that the same argument does not prove that $l \leq l(i)$ because $W_i^{[p]}$ does not necessarily exist.

1.4.3 Proof of Construction 5

This construction defines a set $\mathcal{H}$, consisting of reads and writes of $v^{\bullet}$, that is a higher-level view of a system execution $S, \longrightarrow, \dashrightarrow$ whose operation executions are reads and writes of the two shared registers v, cw and cr. As usual, $\xrightarrow{\bullet}$ and $da*$ denote the induced precedence relations on S that are defined by (3).

Let u denote the shared register v, cw of the algorithm. In this construction, the write $V^{\bullet[k]}$ of $v^{\bullet}$, for $k > 0$, is implemented by the sequence $R \longrightarrow U^{[2k-1]} \longrightarrow U^{[2k]}$, where R is a read of cr and $U^{[i]}$ is the ith write of u. The initial write $V^{\bullet[0]}$ of $v^{\bullet}$ is just the initial write $U^{[0]}$ of u.

Since there is only one reader, the reads of $v^{\bullet}$ are totally ordered by $\xrightarrow{\bullet}$. The ith read S_i of $v^{\bullet}$ consists of the sequence $R_i \longrightarrow CR^{[i]}$ where R_i is the ith read of u and $CR^{[i]}$ is the ith write of cr. For notational convenience, I assume an imaginary read R_0 of u that returns the value $u^{[0]}$, and I define S_0 to be the sequence of operations $R_0 \longrightarrow CR^{[0]}$. The operation S_0 is taken to be the one that sets the initial values of x' and cr'.

The proof of correctness is based upon Proposition 9. Letting $\phi(i)$ denote $\phi(S_i)$, to apply that proposition, it suffices to choose the $\phi(i)$ such that the following three properties hold:

- S_i returns the value $v^{\bullet[\phi(i)]}$.
- If S_i sees $v^{\bullet[l,r]}$ then $l \leq \phi(i) \leq r$.
- If $j < i$ then $\phi(j) \leq \phi(i)$.

I start by defining a function ψ such that R_i returns the value $u^{[\psi(i)]}$ and, if R_i sees $u^{[l,r]}$ then $l \leq \psi(i) \leq r$. Since u is regular, such a ψ exists. Proposition 6 implies:

Z1. If $j < i$ then $\psi(j) \leq \psi(i) - 1$.

By Proposition 7, $U^{[\psi(i)]} \dashrightarrow R_i \dashrightarrow U^{[\psi(i)+1]}$. Suppose $\psi(i) = 2k$. Since $U^{[2k]}$ is part of $V^{\bullet[k]}$, $U^{[2k+1]}$ is part of $V^{\bullet[k+1]}$, and R_i is part of S_i, this implies $V^{\bullet\bullet} \overset{\bullet}{\dashrightarrow} S_i \overset{\bullet}{\dashrightarrow} V^{\bullet[k+1]}$. Hence, property 2 is satisfied if $\phi(i) = k$. Next, suppose that $\psi(i) = 2k - 1$, where $k > 0$. Since $V^{[2k-1]}$ is part of $V^{\bullet[k]}$, we have $V^{\bullet[k]} \overset{\bullet}{\dashrightarrow} S_i \overset{\bullet}{\dashrightarrow} V^{\bullet[k]} \xrightarrow{\bullet} V^{\bullet[k+1]}$, so property 2 is satisfied if $\phi(i) = k - 1$. But we also have $V^{\bullet[k-1]} \xrightarrow{\bullet} V^{\bullet[k]} \overset{\bullet}{\dashrightarrow} R_i$, so property 2 is also satisfied if $\phi(i) = k - 1$. To summarize, property 2 is satisfied by i if the following holds:

Z2. (a) If $\psi(i) = 2k$ then $\phi(i) = k$.
(b) If $\psi(i) = 2k - 1$ then $\phi(i) = k$ or $\phi(i) = k - 1$.

The second statement in the algorithm of Figure 1 consists of nested **if** statements, so executing it executes exactly one innermost **then** or **else** clause. I will use a sequence of **t** (for **then**) and **e** (for **else**) characters to denote such an innermost clause; for example, **tee** denotes the second innermost **else** clause, which is executed if $x_1 \neq x_2$ and $x'_1 = x'_2 = x_2$.

Let a *ttt-read* be one that executes the **ttt** clause of the reader's algorithm, and let a *nice* read be one that is not a ttt-read. The initial read S_0 is defined to be nice. For any $i > 0$, let $\pi(i)$ denote the largest integer such that $\pi(i) < i$ and $S_{\pi(i)}$ is nice. In other words, $S_{\pi(i)}$ is the last nice read before S_i. A ttt-read does not change the value of rtn, x', or cr'. Therefore, when the execution of S_i begins, rtn has the value returned by $S_{\pi(i)}$ and x', cr' has the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$.

I first define $\phi(i)$ inductively for all nice reads, starting with $\phi(0) = 0$. The definition will be made so that Z2 holds for all i. Let i be a nice read, $i > 0$, and assume that properties 1–3 and Z2 hold with $\pi(i)$ substituted for i. In the following discussion, I will refer to the values of variables immediately after the execution of the first statement in the reader's algorithm during the operation execution S_i. Thus, x, cr is the value $u^{[\psi(i)]}$ read by R_i, rtn is the value $v^{\bullet[\phi(\pi(i))]}$ returned by $S_{\phi(i)}$, and x', cr' is the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$.

Consider first the case $\psi(i) = 2k - 1$. In this case, $x_1 = v^{\bullet[k-1]}$ and $x_2 = v^{\bullet[k]}$. If $x_1 \neq x_2$, then properties 1 and Z2 are satisfied only by defining $\phi(i)$ to equal $k - 1$ if S_i returns the value x_1 and to equal k if S_i returns the value x_2. In other words, $\phi(i)$ equals k if S_i executes the **tet** clause and equals $k - 1$ otherwise. Since Z2 is satisfied, property 2 holds.

To prove property 3 for i, it suffices to prove that $\phi(\pi(i)) \leq \phi(i)$, since property 3 is assumed to hold for $\pi(i)$. Property Z1 implies that $\psi(\phi(i)) \leq 2k$, so Z2 implies that $\phi(\pi(i))$ can be greater than $\phi(i)$ only in two cases: (i) $\psi(\pi(i)) = 2k$ and $\phi(i) = k - 1$, or (ii) $\psi(\pi(i)) = 2k - 1$, $\phi(\pi(i)) = k$, and

$\phi(i) = k-1$. But $\psi(\pi(i)) = 2k$ implies that $x'_1 = x'_2 = x_2$, so S_i executes the **tet** clause and $\phi(i) = k$. Hence, case (i) is impossible. If $\psi(\pi(i)) = 2k - 1$ and $\phi(i) = k$, then $x' = x$ and $S_{\phi(i)}$ executes the **tet** clause, so $rtn' = x'_2$. Hence, S_i must also execute the **tet** clause, so $\phi(i) = k$, showing that case (ii) is impossible. This completes the case $\psi(i) = 2k - 1$ and $x_1 \neq x_2$.

If $\psi(i) = 2k - 1$ and $x_1 = x_2$, then I define $\phi(i)$ to be the maximum of $k - 1$ and $\phi(\pi(i))$. Z1 and Z2 (for $\pi(i)$) imply that $\phi(\pi(i)) \leq k$, so this defines $\phi(i)$ to equal either $k - 1$ or k. At this point, I note the following property for later use:

Z3. If $\psi(i) = 2k - 1$, $x_1 = x_2$, and $\phi(i) = k$, then there is a nice read R_j with $j < i$ such that $\psi(j) = 2k$.

The proof of Z3 is by induction on i. The hypothesis, Z1 and Z2 imply that either $\psi(\pi(i)) = 2k$, in which case we can let $j = \pi(i)$, or else $\psi(\pi(i)) = 2k - 1$ and $\phi(\pi(i)) = k$ in which case we apply Z3 with $\pi(i)$ substituted for i.

Returning to the definition of $\phi(i)$, in the case under consideration ($\psi(i) = 2k - 1$ and $x_1 = x_2$), properties 1, 2, and Z2 are satisfied because $\phi(i)$ equals either $k - 1$ or k. Moreover, we obviously have $\phi(\pi(i)) < \phi(i)$, so property 3 is also satisfied. This completes the case $\psi(i) = 2k - 1$ and $x_1 \neq x_2$.

Finally, I consider the case $\psi(i) = 2k$, where $\phi(i)$ must be defined to equal k to satisfy Z2. In this case, $x_1 = x_2 = v^{\bullet[k]}$ and S_i executes the **tte** clause, returning the value x_1. (Since S_i is assumed to be nice, it does not execute the **ttt** clause.) Hence, property 1 is satisfied. Since Z2 holds, property 2 is satisfied. To prove property 3 for i, it suffices to show that $\phi(\pi(i)) \leq \phi(i)$, since the property holds for $\pi(i)$. By Z1, $\psi(\pi(i)) \leq 2k+1$, so $\phi(\pi(i))$ can be greater than $\phi(i)$ only if $\psi(\pi(i)) = 2k+1$ and $\phi(\pi(i)) = k+1$. There are two possibilities to consider: (i) $x'_1 \neq x'_2$ and (ii) $x'_1 = x'_2$. In case (i), $\phi(\pi(i))$ can equal $k + 1$ only if $S_{\pi(i)}$ executes the **tet** clause, which implies that $x'_1 \neq x'_2$ and $rtn = x'_2$; but this is impossible since S_i executes the **tte** clause. In case (ii), Z3 implies that if $\phi(\pi(i)) = k + 1$, then there exists $j < \pi(i)$ with $\psi(j) = 2k + 2$. But Z1 implies that this is impossible, since $j < i$ and $\psi(i) = 2k$. Hence, property 3 holds. This completes the construction of $\phi(i)$ for all nice reads S_i.

To complete the definition of ϕ, if S_i is a ttt-read, I define $\phi(i)$ to equal $\phi(\pi(i))$. Since S_i returns the same value as $S_{\pi(i)}$, property 1 is satisfied. Property 3 obviously holds, since it holds for nice reads and ϕ assigns to every ttt-read the same value as it assigns the most recent nice read. The only thing left to prove is that property 2 holds for a ttt-read S_i. This is perhaps the most subtle proof of the entire paper. It involves proving the remark made earlier, that if a sequence of reads obtains the values (μ, μ), (ν, μ), and (ν, ν), all of the same color, then the last read overlaps the write of (ν, μ).

Let S_i be a ttt-read, and let $(\mu, \mu), c$ be the value $u^{[\psi(i)]}$ read by R_i. Since S_i executes the **ttt** clause, x', cr', which is the value $u^{[\psi(\pi(i))]}$ read by $R_{\pi(i)}$, must equal $(\nu, \mu), c$ for some $\nu \neq \mu$, so $\psi(\pi(i))$ is odd. Let $\psi(\pi(i)) = 2k-1$. Since S_i executes the **ttt** clause, $S_{\pi(i)}$ must return μ, so it must execute the **tet** clause. This implies that $\phi(\pi(i)) = k$, so $\phi(i) = k$, and that the value of cw read by the operation execution $S_{\pi(i)-1}$ must also equal c, so $CR^{[\pi(i)-1]}$ writes the value c. The following operation executions must therefore be performed in sequence by the reader (each one $\longrightarrow$'s the next, but the reader may perform other, intervening operation executions):

- $CR^{[\pi(i)-1]}$: writes $cr[\pi(i) - 1] = c$
- $R_{\pi(i)}$: reads $u^{[2k-1]} = (\nu, \mu), c$
- R_i: reads $u^{[\psi(i)]} = (\nu, \nu), c$
- $CR^{[i]}$: writes $cr^{[i]} = c$

Moreover, the reads between $S_{\pi(i)}$ and S_i also write the value c in cr. Therefore, $cr^{[j]} = c$ for all j with $\pi(i) - 1 \leq j \leq i$. Note also that $\phi(i) = \phi(\pi(i)) = k - 1$.

It follows from Z1 that $\psi(i) \geq 2k-2$. If $\psi(i) = 2k-2$, then Proposition 7 implies that $R_i \dashrightarrow U^{[2k-1]}$. However, that proposition also implies that $U^{[2k-1]} \dashrightarrow R_{\pi(i)}$. Since $U^{[2k-2]} \longrightarrow U^{[2k-1]}$ and $R_{\pi(i)} \longrightarrow R_i$, we see that $U^{[2k-2]} \longrightarrow R_i \dashrightarrow U^{[2k-1]}$. This implies $V^{\bullet[k-1]} \overset{\bullet}{\dashrightarrow} S_i \overset{\bullet}{\dashrightarrow} V^{\bullet[k]}$. Since $\phi(i) = k - 1$, property 2 follows from Proposition 7.

I have shown that $\psi(i) \geq 2k - 2$ and property 2 holds if $\psi(i) = 2k - 2$. To finish the proof, I now show that $\psi(i) = 2k - 2$ by assuming $\psi(i) >$

$2k-2$ and obtaining a contradiction. Since $u^{[2k-1]}$ equals $(\nu, \mu), c$ and $U^{[2k]}$ equals (μ, μ), neither of which equals $u^{[\psi(i)]}$ (because $\mu \neq \nu$), we must have $\psi(i) > 2k$. Let $cr^{[l,r]}$ denote the read of cr in the write of v^* of which $U^{[\psi(i)]}$ is a part. Since $U^{[\psi(i)]}$ sets cw to c, the read $cr^{[l,r]}$ must obtain the value $\neg c$. The writer must therefore perform the following sequence of operation executions, where each $\longrightarrow$'s the next. (There may be other, intervening operation executions.)

- $U^{[2k]}$: writes $u^{[2k]} = (\mu, \mu), c$
- $cr^{[l,r]}$: reads the value $\neg c$
- $U^{[\psi(i)]}$: writes $u^{[\psi(i)]} = (\nu, \nu), c$

By Proposition 7 (and the definition of ψ), $R_{\pi(i)} \dashrightarrow U^{[2k]}$. We therefore have

$$CR^{[\pi(i)-1]} \longrightarrow R_{\pi(i)} \dashrightarrow U^{[2k]} \longrightarrow cr^{[l,r]}$$

so $CR^{[\pi(i)-1]} \longrightarrow cr^{[l,r]}$. By part (b) of Proposition 6, this implies $\pi(i) - 1 \leq l$.

Proposition 7 implies $U^{[\psi(i)]} \dashrightarrow R_i$, so

$$cr^{[l,r]} \longrightarrow U^{[\psi(i)]} \dashrightarrow R_i \longrightarrow CR^{[i]}$$

This implies $cr^{[l,r]} \longrightarrow CR^{[i]}$, so part (a) of Proposition 6 implies $r \leq i$. We therefore have $\psi(i) - 1 \leq l \leq r \leq i$, so regularity of cr implies that $cr^{[l,r]}$ obtains a value $cr^{[j]}$ with $\psi(i) - 1 \leq j \leq i$. However, I already observed that all such values equal c, and $cr^{[l,r]}$ obtains the value $\neg c$. This is the required contradiction, completing the proof.

Part II

The Intersecting Broadcast Machine

2.1 Abstract

This section of the report proposes a new array processor architecture that is

- Effective for arbitrary programs that cannot be mapped onto regular array structures and that, consequently, perform poorly on existing array processors
- Capable of operation in a fault-tolerant mode
- Physically structured to permit high-performance VHSIC implementation.

2.2 Background

There is no need to enumerate the problems for which our current high-performance computers are inadequate; the list would be endless. Moreover, there are many important problems for which our current computers are several orders of magnitude too slow. Remarkable as have been the improvements in computer performance over the past 40 years, there is nonetheless no possibility that the undoubted continued increases in performance will suffice to meet our future needs.

The improved performance of conventional von Neumann computers has been due largely to improved electronic-component technology that allows faster clock cycles and the use of more complex faster circuits, as well as to improved designs that permit operations to be executed in fewer cycles and several operations to be performed concurrently. While further improvements in electronics can be expected, there are very real limitations on the extent to which increased concurrency is possible while still maintaining the von Neumann illusion of purely sequential operation.

Even better performance has been achieved by making the processors more specialized in structure. Two primary examples are the vector processors, such as the Cray and Star computers, which are very effective for processing large matrices uniformly, and the systolic processors, which are very effective for FFT, convolution, and similar signal-processing applications. Unfortunately, there are many applications that do not lend themselves to such specialized processing strategies. For such applications, only parallel processing of the problem by many cooperating processors (whether von Neumann or not) can result in substantially faster processing.

Not only do improvements in electronic-component technology allow the construction of very-high-performance circuitry, but they also permit uniform replication of relatively simple electronic circuits at very low cost. It is clear that the ideal structure for VLSI implementation of a multiprocessor architecture consists of a regular array of processors.

Past experience in using array processors, however, has not been very encouraging. The prototypical Illiac IV computer and the generally similar Intel Hypercube have shown to be effective only when communication within the array is almost entirely between adjacent processors. Communication between arbitrary processors requires that the data be passed via a chain of intermediary processors, which is slow and absorbs an excessive amount of system resources. Even for suitable problems, it has been found to be difficult to map the problem domain onto the array so as to obtain reasonable efficiency; moreover, the approach appears to be almost completely ineffective for less suitable problems.

Another class of array processors is the SIMD (single instruction – multiple data) machines, exemplified by the Connection Machine. SIMD machines are very effective whenever substantially the same sequence of operations must be applied to a large proportion of the cells of the array. The Connection Machine–which, with its one-bit processors, is highly suited to image-processing applications–has been used with great ingenuity in a number of applications. But many applications require a significant proportion of special-case processing and are not implemented efficiently in an SIMD architecture. Furthermore, the problem of communication between arbitrary processors in the array is still significant.

More general are the MIMD (multiple-instruction multiple-data) machines, exemplified by the Butterfly Machine. Such machines contain a set of processors and a set of shared storage modules, the processors accessing the memory through an array of delta switches. The Butterfly Machine also provides a direct link between each processor and its associated storage module. Access to the memory by means of delta switches maintains the appearance of a single uniform memory, equally accessible to all processors, but involves contention among processors for use of the switches, and thus substantially increases memory access time. Consequently, efficient use of the machine requires that most of a processor's memory references be made to its own associated module; this results in allocation problems similar to, but less critical than, those encountered with the Illiac IV type machines.

The problem of mapping an application onto an array is greatly aggravated by the presence of faulty processors. In any large array, it is inevitable that there will be processors that have failed. Allowance must also be made for transient faults, which are much more frequent than solid faults and may cause a significant rate of erroneous results from a large array. The use of VLSI with very small device dimensions, as might be expected in an array implementation, inevitably increases the rate of transient faults.

Errors resulting from faults must be detected and corrected so as to protect the validity of the results. While it is conceivable that adequate protection against solid faults could be provided (at least for batch processing) through some form of periodic testing of the processors, detection of errors resulting from transient faults necessarily requires some form of replication and comparison for all processing operations.

The presence of a faulty processor requires that either

- The error detection and correction algorithms be strong enough to mask the faulty processors continuously (e.g. by majority voting), and that repair be rapid enough to reduce the rate of multiple concurrent faults to an acceptable level, or

- The mapping of the application onto the array be modified to avoid having to use the faulty processor.

Mapping the application onto an array can be quite difficult even in the absence of faulty processors, and is certainly not simplified by introducing irregularities into the array structure. Local adaptation, such as transferring the workload of the faulty processor onto neighboring processors, may overload the processors and increase communication delays. Global adaptation, if possible, will involve moving large amounts of data to accommodate the revised mapping of the application onto the array. The difficulties of adaptive reconfiguration suggest that continuous error detection and correction, which is also effective against transient faults, may be preferable in many circumstances.

Of course, there are some applications in which most of the calculation comprises a search and for which a comparatively short check can be made at the end to confirm the validity of the solution. For such calculations, fault tolerance may be less essential. There are also some applications for which the rate of processor failure may be substantial and, in addition, immediate recovery from error is essential–the most obvious example thereof being the SDI Battle Management System.

In considering an array processor intended for, say, ten years hence, we can reasonably make certain assumptions:

- Main storage will become very inexpensive, and moderate performance processors will become quite inexpensive.
- The major costs and the primary physical constraints will be associated with the interconnection interfaces; the performance of the array will be determined largely by communication costs.
- High-density packaging and interconnection techniques can be applied most effectively if the logical structure of the system corresponds to a feasible physical structure.
- Although individual nodes in the array will be quite reliable, a large array must necessarily contain faulty nodes.

2.3 Objectives

For some applications, a very close match between the structure of the application and the structure of the array processor is not only possible, but offers some advantages from the standpoint of performance. The Intersecting Broadcast Machine is not intended to be competitive for such applications, however.

But there are important applications that do not map easily onto an array and for which the performance of array processors is poor. Our objective is an architecture that performs well for arbitrary applications in which there does not seem to be any preferable mapping onto a regular structure. In the absence of a systematic mapping, the allocation of activities among processors becomes essentially random; thus, the architecture must perform well with such a random allocation and with the consequent random communication.

To ensure reasonable performance, we seek a connectivity structure in which data located randomly within the array can be communicated directly from its source to its point of use without being forwarded through intermediate nodes.

To ensure feasible construction with existing high-density packaging as well as eventual construction on the surface of a wafer, we seek a two-dimensional structure.

To ensure correct operation in the presence of faults, both solid and transient, we seek an architecture that is inherently fault-tolerant.

2.4 Structure of the Intersecting Broadcast Machine

The Intersecting Broadcast Machine consists of two orthogonal sets of buses. Processors are located at the intersections between buses, each processor having two interfaces, one connecting to each of the two buses at that intersection. Thus, for n buses in each set, there are n^2 processors. An arbitration mechanism for each bus allocates that bus among contending processors. The information broadcast on a bus is received and stored by every processor connected to the bus. Processors that will never use the information must still store it, at least temporarily, thus entailing additional storage that is substantial in volume but modest in cost.

Consider a processor, randomly located within the array, that has computed a value, denoted in Fig. 2.1 as A. That processor broadcasts its result

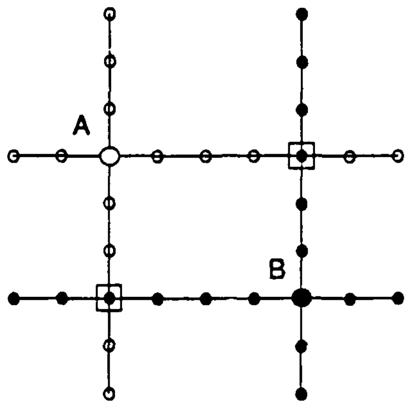


Figure 2.1: The Formation of Intersections. The broadcasting of two results, A and B, from random locations in the array always yields at least two nodes at the intersections of the broadcast, where the next stage of the computation can be executed.

on each of the two buses to which it is connected, and every processor along both buses receives and stores that value. Another processor, also randomly located, computes another value B, which is similarly broadcast over two buses and stored by processors along those buses. There are now two processors, at the intersection of the broadcasts, that have both values and can continue the computation. It should be noted that there was no need to plan or even to know in advance where the results would be computed; the design thus lends itself to complex calculations for which such planning would be difficult

It is anticipated that the array processor will normally be operated in a fault-tolerant mode, as described below. However, it may be appropriate to run some calculations without fault tolerance. We describe such an operation here.

Since each computation is preferably done only once in the array, it is necessary to select one of the two processors at the intersections to perform the computation. This processor can be selected algorithmically, but here

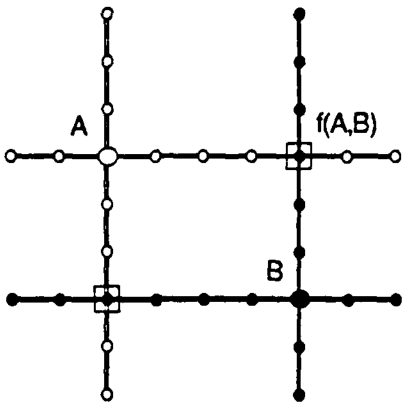


Figure 2.2: The Selection of a Processor by Means of a Race. One of the two processors at the intersections wins the race and broadcasts its results (shown solid). Auxiliary broadcasts (shown broken) inhibit the other processor from broadcasting its results.

we prefer to advocate a race. Each processor enqueues the operation along with other operations it must perform and, when the operation has been completed, the result f(A,B) is broadcast on the two buses to which the processor is connected. As shown in Fig. 2.2, one of the processors will win the race. Its broadcasts not only communicate the result, but also inform the processors that had computed and broadcast the A and B values that the result f(A,B) has been computed. These two processors then generate auxiliary broadcasts on the orthogonal buses. Because the auxiliary broadcasts carry the identity or designator of the result f(A,B) but not its value, they can be much briefer. In the figure the auxiliary broadcasts are denoted by broken lines. These broadcasts inform the fourth processor that the computation has been completed and broadcast, and that there is consequently no need for it to broadcast that result as well.

In the event the auxiliary broadcasts do not reach the fourth processor in time to inhibit its broadcasts, every processor on the four buses will receive both a main and an auxiliary broadcast for the value f(A,B); each processor would then apply an algorithmic selection criterion. It is possible, if not likely, that some of the processors may have already initiated subsequent operations on the basis of a broadcast when the auxiliary broadcast is received that inhibits that broadcast. The processor can readily abandon internal processing, but special action is required to cancel results that have themselves already been broadcast. The technique required, known as a chase protocol, is discussed below in the section on fault tolerance.

The main and auxiliary broadcasts also serve other functions:

- Along each bus there are many processors that have received and stored one of the two values and are waiting for the second value, which they will never receive. The receipt of the main or auxiliary broadcast for f(A,B) indicates that such processors will not be required to compute f(A,B), and therefore can be used to drive the storage management algorithms.
- If the calculation involves the updating of a value and there are several possible updates, the race is not just between the two processors at the intersections but also between concurrent calculations. The broadcasts may indicate which calculation should proceed and which should be restarted with the new updated value. In more complex cases, the race should be to claim a semaphore.

Even though the structure described in this section is not intended to be fault-tolerant, it does exhibit some measure of tolerance for faulty processors. If one of the two processors at the intersections has failed, the other is available to perform the operation.

2.5 Fault Tolerance

The concept is readily extended to provide fault tolerance and, indeed, we do not believe that a large-array processor can be operated effectively in any other mode. To provide fault tolerance, each value is computed and broadcast by two processors in the array. Fig. 2.3 shows the values A and B, each computed and broadcast by pairs of randomly located processors in the array; the broadcasts for B are indicated by broken lines. Note that there are eight nodes at which the values A and B are both available. It would, of course, be inappropriate for the result f(A,B) to be computed and broadcast by all eight; our objective is to have the result f(A,B) broadcast by two nodes, just as the values A and B are.

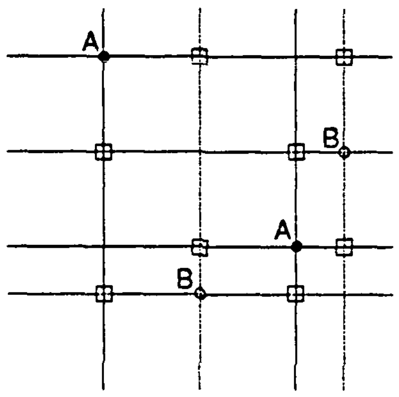


Figure 2.3: The Eight Intersections Resulting from Replicated Broadcasts. When each value is computed and broadcast by two processors in the array, there are eight nodes at the intersections of the broadcasts, where the next stage of the computation can be performed.

Here too it is possible to select the two nodes algorithmically or by means of a race condition. For the latter approach, two alternatives are available, depending on how the auxiliary broadcasts are generated. We describe first the alternative that follows more closely the approach described above for the unreplicated case.

Fig. 2.4 shows that one of the processors has completed the next stage of the computation, denoted by f(A,B), and has broadcast its value on the two orthogonal buses. These broadcasts serve not only to communicate the value, but also to inform two of the other processors in the set of eight

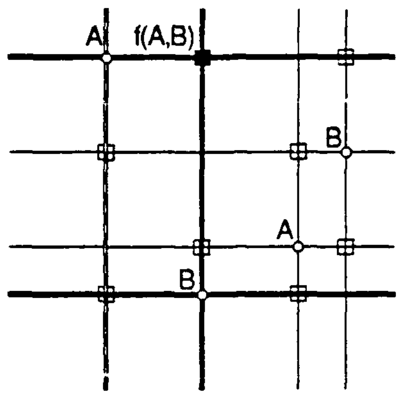


Figure 2.4: The Selection of a Processor by Means of a Race. One of the eight processors at the intersections wins the race and broadcasts its results (shown solid). These broadcasts inhibit two of the other processors, and auxiliary broadcasts (shown broken) inhibit yet another three processors from broadcasting their results. Auxiliary broadcasts are generated by the processors that provide the input values.

that the result has been computed and broadcast. As above, auxiliary broadcasts are generated by the two processors that broadcast the A and B values, shown in the figure by broken lines. The auxiliary broadcasts serve to inhibit three more processors of the set of eight, leaving two processors from that set.

As shown in Fig. 2.5, one of the two remaining processors has computed and broadcast the value f(A,B). Here again two auxiliary broadcasts have been generated by the processors that broadcast the A and B values.

Fig. 2.6 shows an alternative approach to auxiliary broadcasts. Here the set of eight intersection nodes divides into two groups of four each: a group that receives A on a horizontal bus and B on a vertical bus, and a second group that receives A and B in the other directions. We must select one node from each group to continue the computation. The figure shows that one processor has completed the computation of f(A,B) and has broadcast

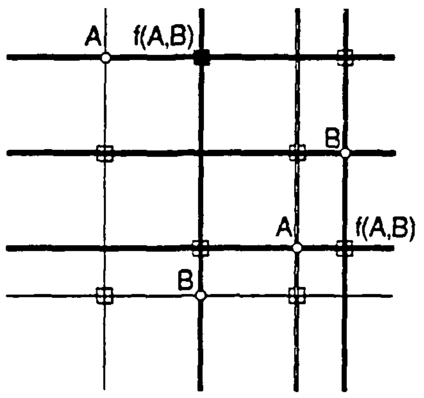


Figure 2.5: The Selection of a Second Processor. One of the two remaining processors wins the race and broadcasts its results. In this case too, auxiliary broadcasts inhibit the other processor.

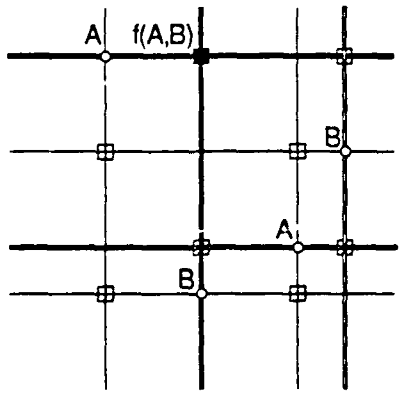


Figure 2.6: The Selection of a Processor by Means of a Race. One of the eight processors at the intersections wins the race and broadcasts its results (shown solid). These broadcasts inhibit two of the other processors. Here auxiliary broadcasts (shown broken) are generated by the inhibited processors and inhibit only one other processor.

its result value on the two buses. These broadcasts inform two of the group of four processors that the result has already been computed and thus to inhibit them from also broadcasting the result. These two processors then generate the auxiliary broadcasts. This differs from the foregoing approach, in which the auxiliary broadcasts were generated by the processors that had provided the A and B values. The auxiliary broadcasts indicate to the fourth processor of the group that the result has been computed and broadcast.

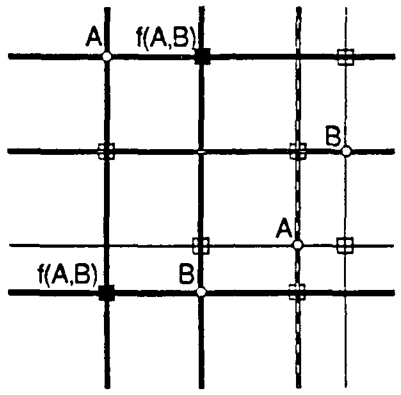


Figure 2.7: The Selection of a Second Processor. The second group of four processors is handled similarly. The main broadcasts, done by the processor winning the race, inhibit two of the processors, while the auxiliary broadcasts inhibit the fourth.

The second group of four processors is handled similarly, as shown in Fig. 2.7.

In both of these alternatives, we started with the computation and broadcastingof the values A and B by two processors in the array; now the next computation is similarly performed and broadcast by two processors, as shown in Fig. 2.8. Note that for each alternative, on each bus carrying a broadcast value, there are three processors that hold independent versions of that value:

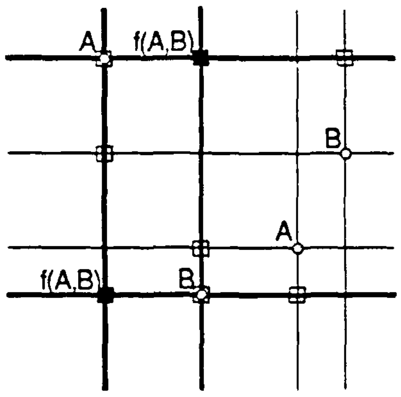


Figure 2.8: Fault Tolerance. Initially the A and B values are each broadcast by two processors in the array and, subsequently, f(A,B) is also broadcast by two processors. On each bus there are three processors with independently computed values, thereby allowing majority voting if necessary.

- The processor that computed and broadcast the value
- Another processor that can compute the result but lost the race
- A processor that cannot compute the result but received it on an orthogonal bus.

Consider the upper horizontal bus in Fig. 2.8. There is a processor that has computed and broadcast the value f(A,B), while to its right another processor has computed the result but was inhibited by prior broadcast of that result. To the left is a processor that has not computed the result but received the value f(A,B) from the broadcast by the selected processor of the second group of four. Any difference between the broadcast value and these alternative values results in the latter's also being broadcast; every processor on the bus then has three independently computed values, among which it can choose by majority voting.

Unfortunately, this majority vote occurs after the first, possibly erroneous value has been received by all of the processors along the bus. If the majority vote indicates that the first value is indeed erroneous and provides another, different value, it is possible that some of the processors may have commenced further operations based on the erroneous value and may even have broadcast the results of such operations. It is clearly essential that these erroneous results be retracted as rapidly as possible. This problem is not unique to the Intersecting Broadcast Machine, but arises in almost all fault-tolerant distributed systems. It has been investigated by Randell and Merlin[1], who refer to it as the chase protocol problem, and by Liskov et al.[2], who call it orphan detection. The chase protocol is so named because the retraction message must chase after erroneous results, possibly through several intermediate processors. Of interest with regard to these algorithms is the question of convergence, namely, whether the chase after erroneous values ever actually terminates. Convergence depends on the ratio of the time for computation and transmission of new result values to the time it takes to propagate the chase messages. Provision may be made to give chase messages priority over other broadcasts, so as to improve convergence of the protocol.

The first of the alternatives, in which the auxiliary broadcasts are generated by the processors that supply the input values, matches closely the approach required for unreplicated operation. However, the second alternative, in which the auxiliary broadcasts are generated by processors at intersections, has better fault tolerance properties. In particular, as regards the second alternative, if one of the values is broadcast by only one node, while the other is broadcast by two, the result of the operation is broadcast by two nodes. Of course, all three independently computed values are not available on each bus for majority voting, if so required. Some buses may have only two values available, allowing error detection but not correction. Consequently, the second alternative permits recovery from a situation in which, for whatever reason, a value is broadcast by only one node of the network. The first alternative does not possess this property.

We must consider not only the possibility of processor failure, but also bus failure. First, we note that solid failure of a bus prevents all of the nodes along it from receiving any further broadcasts on that bus. Consequently, since that entire row of processors can no longer be at an intersection, they are therefore essentially lost to the system. Thus, in view of the very detrimental effect on system performance, the design should attempt to minimize the rate of bus failure.

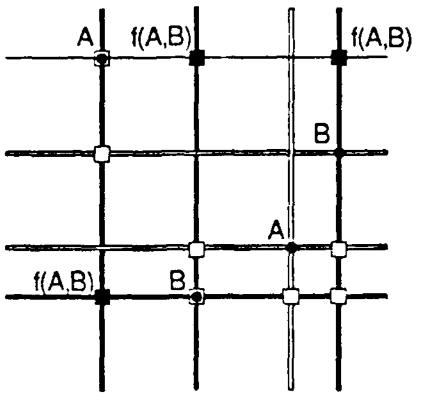


Figure 2.9: The Effect of Bus Failure. Some broadcasts do not take place, and some of the processors may not be inhibited. There may be three broadcasts of a value in one direction, but only a single broadcast in the other direction.

Immediately after a bus failure, the processors along the bus may still be executing operations based on values received prior to failure of the bus. The effect of bus failure on such operations is shown in Fig. 2.9. If one of the processors connected to the faulty bus wins the race in its group of four, it will be able to broadcast its result on the working bus but not on the one that has failed. Consequently, the other processor of the group of four that is attached to the faulty bus will not receive the broadcast and will not know that it should refrain from broadcasting its result value. Thus, as is shown in the figure, the value may be broadcast on three buses in one direction, but on just one bus in the other.

Fig. 2.10 shows the possible next computation, with the values renamed for convenience. The value C is broadcast on three vertical buses and one horizontal bus, while D is broadcast on two buses in each direction.

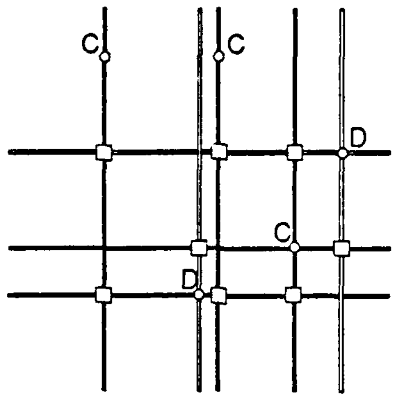


Figure 2.10: Continued Computation after Bus Failure. To examine the effects of continued computation after bus failure, we consider a computation in which one of the arguments has the three/one-broadcast pattern.

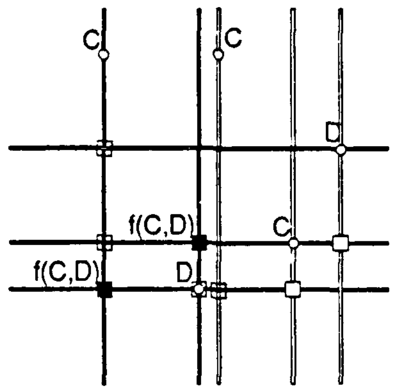


Figure 2.11: Recovery from Bus Failure. Starting with one argument in the three/one pattern, the next result is broadcast by two processors and carried on two buses in each direction.

Fig. 2.11 shows the result of the next computation. The value f(C,D) is broadcast on two horizontal and two vertical buses, as desired. On three of these buses, at least three independently computed values are available for majority voting if necessary. But there may be one bus on which only two values are available. Thus, during recovery from bus failure, some buses may transiently only be able to provide error detection but not error correction.

2.6 Bus Structure

The architecture is built largely around the bus structure. The buses might be quite similar in design to those employed by such companies as EXLSI, Sequent, Encore and Alliant. Such buses, which can accommodate up to 30 interfaces, currently run at about 40 MHz with 32 or 64 data signals. 100 MHz or more may soon be possible, and optical versions of such a bus should be able to operate at even higher rates. Transfers across the bus must carry both an identity for the value and the value itself, which might be as small as a single word or could be quite large.

Since the buses are of course contention buses, arbitration circuitry is required to allocate their use to the processors. This circuitry can be separate from the bus itself and need impair performance.

The regular two-dimensional bus structure has physical characteristics that are suitable for mass production and permit the design of very-high-performance buses. In particular, a two-dimensional structure is very appropriate–perhaps even essential–if the array processor is to be implemented directly across a single wafer, as may be possible in the future.

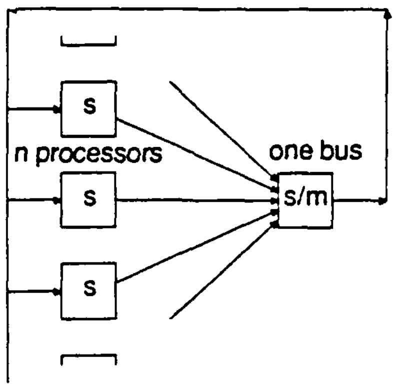
2.7 Performance Model

Since all the processors in a row or column must share the same bus, there is contention for use of the buses; access to them therefore can become a limiting factor in the design. As the array is made larger, with constant processor and bus performance, there will necessarily come a point at which the buses become overloaded. When this occurs, further increases in array size will add little to overall performance. Hence, the design scales well only up to the point at which the buses become saturated.

The number of processors that can share a bus without saturating it depends on (1) the ratio of the performance of the processors to the performance of the bus, (2) the length of the typical computation, and (3) the size of the typical result to be transmitted across the bus. If the typical computation is very short, perhaps a single operation as in a dataflow machine, it can be expected that the time to perform an operation will be less than or equal to the time to transmit the result (probably a single word). With many processors competing for the bus, saturation is clearly inevitable.

But a coarser granularity of computation should increase the duration of the computation by more than it increases the size of the results, particularly if the computations and data structures are carefully chosen. Current designs for fast buses, such as will be necessary here, can accommodate up to 30 interfaces on the bus. Consequently, a good objective would be to find programs in which the typical program fragment takes about 30 times as long to execute as the bus needs to transmit the result.

A simple queueing theory model has been constructed to determine the effect of the ratio of processing time to bus transfer time. The model, as shown in Fig. 2.12, considers one column (or row) of processors together with the bus that serves that column.



F.gure 2.12: The Queueing Theory Performance Model. The model considers one column of n processors and the bus that serves that column.

Let	n	be the number of processors in a row or column
	s	the mean time for a processor to process an operation (excluding time in queues waiting for operands or the processor)
	e	the mean utilization of a processor
	m	the ratio of bus speed to processor speed,
giving	s/m	as the mean time to transmit a result over the bus.

Since the utilization of a processor is e, by elementary queueing theory, the mean number of operations being performed or waiting in the queue to be performed by that processor is $\frac{e}{1-e}$.

Since n processors make requests on the bus and since the bus is m times as fast as a processor, the utilization of the bus is ne/m. Consequently, the mean number of operations being broadcast or queued, waiting to be broadcast, is

$$\frac{ne}{m(m-ne)}.$$

There are n^2 processors in the system and n buses (we do not count the orthogonal set of buses, since each broadcast must use one bus from each set. Thus the number of concurrent operations to achieve the processor utilization of e is

$$\frac{n^2 e}{1-e} + \frac{n^2 e}{m(m-ne)}.$$

Fig. 2.13 shows the results of this queueing model for a system with $n = 30$. When $m = 30$, high utilization of the processors requires many more concurrent operations than processors, but acceptable utilization is obtained when the number of concurrent operations is equal to the number of processors. For a slightly slower bus, with $m = 20$, there is a loss of processor utilization resulting from bus saturation when there are a great many concurrent operations. But, if the number of concurrent operations is comparable to the number of processors, the loss of processor utilization is not substantial. Significantly slower buses, as when $m = 10$, become saturated before adequate processor utilization is achieved. There is no benefit from using much faster buses; the curve for $m = 50$ is indistinguishable from that for $m = 30$.

2.8 Load Balancing

The queueing theory model above makes an assumption that the load is spread across the array uniformly and randomly. But how can we be sure, even if we start with a uniform and random distribution, that execution of the program will not tend to cluster much of the load onto a few processors, leaving others underutilized. We consider a simple stochastic model of the system:

Let p_i be the proportion of the load on the ith horizontal bus
q_j the proportion of the load on the jth vertical bus
r_{ij} the proportion of the load on processor i, j

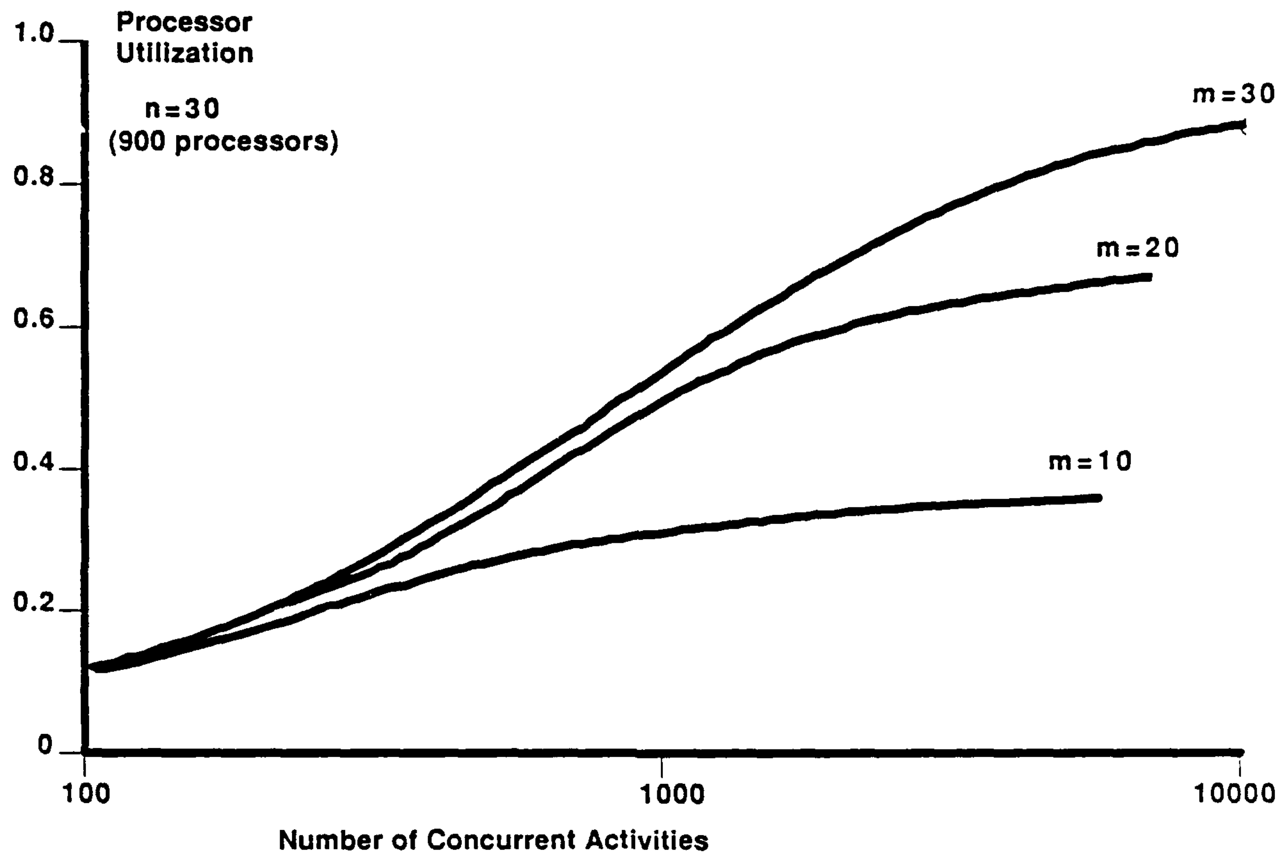


Figure 2.13: Processor Utilization. Processor utilization depends on the number of concurrent operations as well as on the ratio m of bus to processor speed. The figure is drawn for a system of 900 processors and two sets of 30 buses.

We can relate these by

(1) $\sum_i p_i = \sum_j q_j = \sum_{i,j} r_{ij} = 1$

(2) $p_i = \sum_j r_{ij}$

(3) $q_j = \sum_i r_{ij}$

(4) $r_{ij} = kp_i q_j$

The first of these equations is clearly valid, while the second and third depend on the assumption that results generated by different processors are drawn from the same size of distribution, which is not an unreasonable premise. Equation (4) is open to some doubt, for the program does not select operands for processing entirely at random and thus can distort the distribution. However, even if we assume the validity of Equation (4), it is evident that these equations have no unique solution. Thus there can be no expectation that the load will remain uniformly distributed, nor, in particular, any expectation that the load will be self-stabilizing.

But all is not yet lost. Consider a system in which the probability of a processor's performing an operation depends on that processor's current load.

(1) $\sum_i p_i = \sum_j q_j = \sum_{i,j} r_{ij} = 1$

(2) $p_i = \sum_j r_{ij}$

(3) $q_j = \sum_i r_{ij}$

(4) $r_{ij} = kp_i q_j \times r_{ij}^{-b}$ or $r_{ij} = 0$, where $b > 0$ ($b = 0$ is the previous case)

or (4′) $r_{ij} = (kp_i q_j)^{\frac{1}{1+b}}$.

The revised version of (4) contains a factor r_{ij}^{-b} that reduces the probability that a heavily loaded processor will undertake an operation and, conversely, increases the probability that a lightly loaded processor will do so.

Now $p_i = \sum_j (kp_iq_j)^{\frac{1}{1+b}} = k^{\frac{1}{1+b}} p_i^{\frac{1}{1+b}} \sum_j q_j^{\frac{1}{1+b}}$

and $p_i^{\frac{b}{1+b}} = k^{\frac{1}{1+b}} \sum_j q_j^{\frac{1}{1+b}}$ if $p_i \neq 0$,

showing that p_i is independent of i.

Thus $p_i = q_j = \frac{1}{n}$, $r_{ij} = \frac{1}{n^2}$, and $k = \frac{1}{n^{2b}}$.

We do not expect that, in practice, the actual form of the term describing the probability of a processor's performing an operation will be precisely as presented here; this form was chosen to facilitate analysis, with the objective being merely to show that negative feedback can indeed stabilize the load. The race algorithms described above provide this feedback.

Not only should the algorithms of the system spread the load uniformly, but they should also be stable. Unfortunately, since a more detailed analysis involves both queueing theory and control theory, it is quite difficult. For the present, we note that system software usually appears to be heavily damped and seldom becomes unstable.

One of the most significant differences between the Intersecting Broadcast Machine and other array processor architectures is now apparent. Other architectures require a particular geometric mapping of the application onto the array in order to obtain minimal communication and high performance. The Intersecting Broadcast Machine operates on the basis of a random allocation of the application to the processors of the array, and performs acceptably for that random allocation. There are many applications for which the particular mapping is hard to find or does not exist, and so the resulting performance may be very bad. But all applications can be allocated at random and should perform acceptably on the Intersecting Broadcast Machine. Furthermore, a particular mapping may be seriously disrupted by faulty processors, while a random allocation should not be affected significantly. Consequently, the Intersecting Broadcast Machine should be more general and more robust than other array processor architectures.

2.9 Programming

In some ways the architecture resembles a data-driven dataflow machine. Most of what has been learned about dataflow architectures is applicable, especially as regards the naming of values. However, there are significant differences. In a dataflow machine, the selection of data to be processed is determined primarily by the dataflow program; it is largely independent of the values of the data. The Intersecting Broadcast Machine, in contrast, is perhaps most effective when the selection of data items to be processed together depends on the values of the data; for this reason, the allocation of data to processors cannot be preplanned.

Dataflow architectures are usually purely functional, whereas it is possible to operate the Intersecting Broadcast Machine as an imperative machine—while taking precautions, of course, to avoid unintended interactions between concurrent operations.

Like many other dataflow structures, operations are necessarily monadic or dyadic (one or two inputs). However, it is possible to show that, if the n values are randomly distributed across the array, the number of broadcasts necessary to collect all n values at one node is not increased by gathering them in pairs.

Because our objective is to reduce communication costs, it would appear that a relatively coarse granularity of computation will be appropriate. Research is currently in progress at the University of Illinois and elsewhere on automatic decomposition of conventional programs into fragments of an appropriate granularity that can be executed in parallel. This research is essential to the effective operation of the University of Illinois Cedar supercomputer; it promises to be equally effective for the Intersecting Broadcast Machine. As yet, no substantive results have been reported.

2.10 Applications

There are many applications for which the Intersecting Broadcast Machine is no better than other array processors. Where the geometric structure of the application matches closely the structure of the array processor, communication among processors can be efficient and the overall performance of the array processor can be very good. Typical applications of this type are image and signal processing and the solving of partial differential equations. But even programs, whose inner loops perform regular calculations suitable for array processing, may have substantial sections of initialization and analysis that are not so regularly structured and efficiently processed. In some cases, the inefficiency of processing these unstructured portions of a program are such that they dominate overall processing time.

The Intersecting Broadcast Machine should be capable of running a wider range of programs than some other array processors, since it is not dependent on finding a good mapping of an application onto the array. Many programs have a very complex structure that is often not well understood; this is especially true of command and control programs and AI programs.

An example of an application that appears to be ideal for the Intersecting Broadcast Machine is that of discrete Monte Carlo particle dynamics. This application is very important to Lawrence Livermore Laboratory because it allows the modeling of very violent events that are not well modeled by the fluid dynamic approximations used for events that are less violent. Discrete particle dynamic simulations track each particle individually and model the interactions among particles. If the event is sufficiently violent, the spatial relationships among particles can differ substantially between time steps, so that it is continuously necessary to reascertain which particles are close enough to one another to interact. The nature of the interaction calculations depend on the separation, velocities, and types of the interacting particles and, in addition, may involve a number of different code sequences; each particle may interact with only a few or with many other particles, depending on its situation. It has been found difficult to vectorize this calculation, and the differences in the calculations for each individual particle make an SIMD approach less effective. The huge amount of calculation required, however, clearly indicates the need for an array processor. From a superficial standpoint, the Intersecting Broadcast Machine appears to be quite suitable.

Rather similar calculations that might also be very appropriate include the problem of conflict prediction in air traffic control conflict prediction problem and ray tracing for three-dimensional computer graphics.

Another application for which the Intersecting Broadcast Machine appears to be quite suitable is SDI Battle Management. Here again the matching of information from diverse sources and the dynamic allocation of battle resources on the basis of complex optimization criteria may well be beyond the computational abilities of sequential processors. Furthermore, it is not easy to implement these procedures on conventional array processors. The battle management software is likely to be changed and elaborated rather more frequently than is necessary for other types of applications. Such modifications may be difficult to make if the design of the software has to be tied to a specific mapping of the application onto the array processor; an architecture that also allows the data structures to be readily modified is more suitable. The battle management application is one for which fault tolerance is clearly imperative.

2.11 Conclusions

The Intersecting Broadcast Machine array processor architecture is currently only an interesting idea that shows promise of being

- Effective for arbitrary programs that cannot be mapped onto regular array structures and that, consequently, perform poorly on existing array processors
- Capable of operation in a fault-tolerant mode
- Physically structured to permit high-performance VHSIC implementation.

There is still much to be done before we can be confident that the architecture will indeed perform as envisaged. It is necessary to investigate, in particular,

- More details of the design, including
  - Broadcast protocols
  - Naming of result values
  - Representation of programs
  - Recognition of intersections at which operations must be performed
- Methods of programming the architecture
- Studies of sample applications.

Part III

Broadcast Protocols for Distributed Systems

3.1 Abstract

This section of the report proposes a novel reliable broadcast protocol for the link level of the protocol hierarchy. The protocol exploits the broadcast nature of the physical communication media typically used in local area networks. A combination of positive and negative acknowledgment strategies allows reliable operation without requiring a separate acknowledgment from every recipient of a message. This work was undertaken in collaboration with Professor L. E. Moser of California State University, Hayward.

3.2 Introduction

Many distributed computer systems use a communication mechanism that is physically a broadcast medium, such as the Ethernet or a packet radio system. Other common communication media, such as the token ring, could function as broadcast media, even though they are not normally so used. The advantage of a broadcast communication medium is that it makes it physically possible to distribute a message simultaneously to several destinations.

There are important activities in a distributed computer system that involve many processors simultaneously and that would benefit from broadcast communication. Among these are scheduling and load balancing, synchronization, access to distributed information, update and commit for distributed databases, and transaction logging.

Existing communication protocols do not allow distributed computer systems to make use of this broadcast capability, but rather require all messages to be point-to-point, from a single source to a single destination. If the nature of the application is such that broadcast communication is appropriate, existing systems must send many individual messages and receive corresponding individual acknowledgments. In a network of N nodes, this results in a total of 2N messages, when perhaps a single broadcast message might have sufficed. The high cost of broadcast communication is not only wasteful of the communication resource, but it also limits the size of the distributed system by saturating the communication system and discourages the use of truly distributed algorithms because of their unnecessarily high communication cost.

Reliable transmission of a message requires the ability to retransmit the message because of damage or loss in transit. Within the ISO protocol hierarchy, the primary responsibility for ensuring this reliable transmission across the broadcast communication medium lies with the link-level communication protocol. This protocol is directed towards that level of the hierarchy. Consequently, the protocol provides only services appropriate to the link level, in contrast to other atomic broadcast protocols that ignore the hierarchy and are designed to be entirely self-contained. For example, our protocol can determine whether a node has acknowledged receiving a message, but has no responsibility for network membership or network reconfiguration following a failure. Some of what we describe below may also be relevant to other levels in the protocol hierarchy, particularly the transport level that ensures reliable transmission between hosts.

Most existing link-level protocols use positive acknowledgments, in which the recipient of a message explicitly transmits an acknowledgment of its receipt, either as a separate message or as part of another message. The sender of the original message uses a timeout to trigger retransmission if no acknowledgment is received from the recipient. In a broadcast context, such protocols require individual acknowledgments from each recipient, even if it is possible (which it usually is not) to take advantage of the broadcast medium to disseminate the initial message to all recipients. Thus, broadcasting with positive acknowledgments could reduce the number of messages from 2N to N+1, which is still far from taking full advantage of the broadcast medium.

To eliminate this overhead, we must use a negative-acknowledgment strategy, in which most nodes transmit no acknowledgment if they receive a message successfully, but rather transmit a negative acknowledgment if they become aware that they have not received a message.

We should also note that realistic systems will contain many semi-independent processes within each node. The overall communication system may need to deliver the broadcast message to several such processes, but such delivery is not the responsibility of the ISO link-level protocol. We do not consider further multiple delivery within a node.

The negative-acknowledgment broadcast protocol described here does involve costs, particularly computation costs that might in many cases be borne by an interface microprocessor. There are also delay costs that must be compared with the delays caused by the heavier communication load of existing protocols and algorithms. The utility of such protocols also depends on some assumptions:

> The performance of individual processors and the demand for communication generated by such powerful local computation will outstrip the available bandwidth of the communication medium.

- Many applications will require distributed computation and consistent distributed data spread across a local distributed system.
- Requirements for consistency with remote sites (beyond the broadcast communication medium) will be minimized.

There is a possibility that continued progress in communication technology, such as 100 MHz fiber links, will eliminate any communication bottlenecks and eliminate the need for more efficient broadcast protocols. But techniques applicable to communications are also effective in enhancing the performance of processing nodes. It is possible, even likely, that advances in semiconductor technology will allow much greater increases in processor performance, in that an entire processor can be contained in a small controlled package, while interprocessor communication will be subject to gross physical constraints. Consequently, we anticipate that the communication medium will continue to be a limiting resource in distributed systems and that broadcast protocols will become an important technique for distributed systems.

Given a reliable and efficient broadcast protocol, it then would be possible to take advantage of it to construct efficient distributed application algorithms. We have started to investigate such algorithms for distributed mutual exclusion, locking, and synchronization, as well as for update and commit in a distributed database.

3.3 Existing Protocols

The most detailed existing description of a reliable broadcast protocol is that by Chang and Maxemchuk[3]. Their protocol requires that all messages pass through an intermediary node, called the token site. A node wishing to broadcast a message must communicate it to the token site, using a positive-acknowledgment protocol. Using a negative acknowledgment protocol, the token site then broadcasts the message to all recipients; any missing messages are detected by gaps in the sequence. The use of a single common intermediary makes the negative-acknowledgment technique more effective. A complex token passing protocol is used to detect failures at the token site, to select a new token site, and to retransmit messages affected by the failure. Although two messages and one acknowledgment are required for every message broadcast in the absence of errors, the token passing protocol can, in fact, add significantly to the number of messages if transmission errors are frequent.

Schneider has described a reliable broadcast protocol capable of operating on partially connected networks[4]. His protocol can operate in a more complex network structure with gateways, but does not have high efficiency on a local network. It might approriately be implemented using the protocol described here at the local level.

Several authors[5,6] have described broadcast protocols in which each message is followed by an empty pause or a null message for a token ring. A node that detects the presence of the message, but is unable to receive it uncorrupted, transmits a negative acknowledgment in the pause or null message. Such algorithms are effective against reception faults, but not against transmission faults, momentary network-partitioning faults, or processor fail-stop faults.

A further class of broadcast protocols–asynchronous atomic broadcast protocols[7]–is more concerned with maintaining completely global consistency of message ordering and delivery in the presence of node failures (the Chang and Maxemchuk protocol also provides asynchronous atomic broadcasts). Such protocols necessarily involve maintenance of a configuration of currently operating nodes and mechanisms for reconfiguration in the event of node failure. These features are not appropriate to the link level of the hierarchy, but are more appropriate to the network, transport, and higher levels of the hierarchy. They can be built on top of the reliable link-level broadcast provided by our protocol.

3.4 Requirements and Objectives

Our objective is to provide a reliable link-level or transport-level protocol. Messages should be capable of being broadcast simultaneously to many destinations, without the need for explicit acknowledgment by every recipient. The originator of the message should be assured that all working destinations have received the message, or that one or more destinations did not receive the message and that it should therefore be retransmitted. It should also be possible to confirm that certain specified destinations were working and did receive the message.

The protocol must also be able to ensure that messages from one source will be delivered in the order in which they were originated by that source. Since some messages may have to be retransmitted to compensate for errors, this may require the use of sequence numbers to reorder the messages after reception. There is no requirement that messages from different sources be received in any particular order.

Reliable communication that depends on backward error recovery and retransmission necessarily incurs a delay before the originator of a message can be certain that all the intended recipients of the message have indeed received it. In a positive-acknowledgment system, that delay is represented by the time until the last acknowledgment is received. In a negative acknowledgment system, the situation is more complicated. Some kinds of messages are such that it is the time to the first response that is important (e.g., "Give me the value of X"). For other types of messages (e.g., "Update the current value of X"), the delay may be the time until the originator can be certain that every working node has received the message. This time is much less certain in a negative-acknowledgment system, but may be an important performance parameter in some contexts. The performance measures for the protocol must therefore be

- The load placed on the communication medium
- The load that causes the medium to become saturated

- The delay incurred until the originator can be certain of delivery.

Generally, the load imposed in the absence of errors is more important than the additional load induced when errors occur, since they are not very frequent. Similarly, the delay until confirmation of delivery is usually more important for delivery to a working node. Deduction that a node has failed may be based in part on information provided by this level of the hierarchy to the effect that no response has been obtained from the node; the decision, however, lies above the link level.

The protocol must operate reliably in a network subject to a variety of faults. Among these, in particular, are the following:

- Transmission faults, in which the transmitted message is either not received by any destination or is received in a garbled condition by all destinations. We assume that transmission faults are relatively infrequent.

- Reception faults, in which one or more destinations do not receive the message or receive it garbled, while other destinations receive it correctly. Again we assume that reception faults are relatively infrequent, say, substantially fewer than one error per N messages in an N node system.

- Network-partitioning faults, in which the network is divided by the fault into two or more subnets, with communication remaining possible within each subnet but not among subnets.

- Node fail-stop faults, in which a node ceases operation. We assume that a failed node rejoining the network is aware that it has failed, since transitional acknowledgment rules must be applied.

We assume that a node can apply adequate checks to a message it has received to ensure that it has been received uncorrupted. We exclude faults involving babbling nodes and faults resulting in the total inability of all nodes to use the communication medium, since there is no way of ensuring recovery from such faults with a single communication medium. Malicious nodes are excluded. We also exclude faults that result in one or more pairs of nodes being systematically unable to communicate, even though there are other nodes with which both members of a pair can communicate. Such faults could result from misadjusted transmitters and receivers that are marginally operative and thus able to communicate with some, but not all, other nodes. We exclude this type of fault because it does not lie within the scope of a link-level protocol; we do not wish to complicate the protocol with a forwarding requirement that is properly the responsibility of the network level.

3.5 The Broadcast Protocol

Expressed in informal terms, the proposed broadcast link-level protocol requires that

- Each message be broadcast with a header in which there is a message identifier containing the source of the message and a message sequence number. A version number is also included in the identifier to distinguish retransmissions. Sequence numbers can be repeated over some suitably long interval. The message also carries an error-detecting code. Other fields of the header, such as a message destination list, may be present but do not play any part in this protocol.

- Each node maintains a list of positive- and negative-acknowledgment message identifiers. Whenever it broadcasts a message, it appends this list of acknowledgments to the message and then clears its list.

- When a node receives a message not previously received in an uncorrupted state, it adds the identifier as an acknowledgment to its list. If the message is uncorrupted, the identifier is added as a positive acknowledgment; if the message is corrupted but with an uncorrupted header, the identifier is added as a negative acknowledgment.

- When a node sees a positive acknowledgment appended to a message that it receives, it deletes from its own list any positive acknowledgment for that message. When it sees a negative acknowledgment for a message, it deletes from its list any acknowledgment for that message, whether positive or negative.

- When a node sees a positive acknowledgment for a message that it has not received, it adds a negative acknowledgment to its list.

- If a node has no messages pending, it may be necessary to construct a null message to carry acknowledgment messages. The acceptable delay before transmitting a null message may differ for positive and negative acknowledgments.

- When a node receives a negative acknowledgment for one of its messages, or has received no positive acknowledgment within some time interval, it retransmits the message. The retransmission must be identical to the prior transmission, and thus must carry with it all of the acknowledgments, positive or negative, carried by the prior transmission of that message.

As an example, consider the following message sequence, in which upper-case letters represent messages (we do not bother to denote the source of the message directly), lower-case letters represent acknowledgments, and overhead bars denote negative acknowledgments.

$$\mathrm{A} \qquad \mathrm{Ba} \qquad \mathrm{Cb} \qquad \mathrm{D}\bar{\mathrm{c}} \qquad \mathrm{Cbd} \qquad \mathrm{Ec}$$

Here the negative acknowledgment of C that accompanies message D triggers a retransmission. Note that the node broadcasting message E also acknowledges message C; in doing so, it implicitly acknowledges messages B and D and through B message A as well. This implicit acknowledgment is the basis of the reliability property described below.

The effect of missing several messages can be considered in this example.

A   Ba   Cb   D$\bar{c}$ Cbd   Ec$\bar{b}$ Bae   Fb

Here the node broadcasting message D received message C garbled and saw nothing of message B. When C is retransmitted with a positive acknowledgment for B, that node becomes aware that it missed B and transmits a negative acknowledgment. Thus a short sequence of missing messages can be recovered; however, it would be unwise to depend on this technique for recovery from a lengthy node failure.

3.5.1 Notes

Depending on the format of messages and the form of error-detecting codes used, it may not be possible to determine with confidence the identifier of a message that is received corrupted. If so, nodes that receive such corrupted messages cannot enqueue a negative acknowledgment for fear that the identifier might be incorrect, but instead must simply ignore the corrupted message. If some other node has received the message uncorrupted and broadcasts an acknowledgement, then one or more of the nodes that received the message corrupted will generate the negative acknowledgment, based on the positive acknowledgment for a message that they have not yet seen. If no node receives the message uncorrupted, no positive acknowledgment will be generated and the originating node will retransmit the message after the timeout. Because of the nature of the acknowledgment protocol, the timeout need not be long and thus the effect on performance should be negligible.

It is permissible but not essential for a node to broadcast a positive acknowledgment for a message that it had already received uncorrupted. Nodes should not broadcast negative acknowledgments for such messages, as this can cause additional, unnecessary retransmissions, possibly never terminating if errors are sufficiently frequent.

Because a retransmission must be identical to each previous transmission of the same message, a node that receives a message carrying a negative acknowledgment of one of its own messages must not append the positive acknowledgment of that message to the retransmission; the positive acknowledgment must wait in the queue for some subsequent message. Permitting further acknowledgments to be added to a message on retransmission would preclude a node that has already received the message from ignoring the retransmission, and would thus risk incurring the nonterminating sequence of retransmissions.

When a node joins or rejoins an already operating network, the first few positive acknowledgments that it receives will be for messages that were broadcast prior to its entry into the network and that it therefore has not received. If the node broadcasts negative acknowledgments for those messages, forcing their retransmission, it will receive with those messages the positive acknowledgments to even earlier messages. This results in replaying the entire message history of the network in reverse order! To avoid this, we require that a processor joining or rejoining the network should broadcast negative acknowledgments only for messages with sequence numbers greater than the sequence number of a message that it has received correctly.

The description of the protocol given above is a rather operational description that requires immediate performance of the operations, without regard to other node performance constraints or the need to make continuous use of the broadcast medium. Clearly, the performance of the protocol is improved if each node can respond very rapidly to each message it receives. Ideally, on seeing an acknowledgment appended to a message, the node should be able to ensure that it will not also transmit the same acknowledgment, even if it is next in line to transmit a message that has already been prepared with that same acknowledgment attached. Similarly, on receiving a message, a node might be able to include the acknowledgment with its next message, even if the latter must be transmitted immediately.

In practice, however, it takes time to check the cyclic redundancy check code, manipulate acknowledgment queues, and construct message packets, while efficient use of the communication medium requires that the next message be transmitted with as little delay as possible. The idealized expectation that reception of a message can be reflected in the acknowledgments that accompany the next message is unrealistic. Nevertheless, delays in broadcasting acknowledgments and extra acknowledgments, either positive or negative, have no logical effect on the protocol and only a very small effect on performance. Thus it is of little significance if processing constraints do not permit immediate acknowledgment or immediate removal of acknowledgments from pending messages, so that some acknowledgments are delayed a few messages while others are broadcast twice. The formal temporal logic specifications impose temporal constraints on acknowledgments that do not imply the unrealistic requirements inherent in the behavioral description.

3.6 Reliability Property

Provided that the proportion of messages received corrupted is much less than 1/N for an N node network and that there are no pairs of nodes that are systematically unable to communicate, the protocol appears quite robust. We can define for it a strong reliability property:

> When a node acknowledges a message, if there are no unacknowledged messages prior to that message and if no prior message has an outstanding negative acknowledgment, then the node must have received correctly every message prior to the message it acknowledged.

The proof of this property is based on the representation of messages and acknowledgments as a finite directed acyclic graph G_A. Nodes of the graph represent messages, while it edges represent positive acknowledgments. We use the term *graph node* to denote the nodes of the graph, so as to distinguish them from network nodes. The construction of the graph G_A is as follows:

- Transmission (or retransmission) of a message M adds a graph node M to G_A
- Transmission (or retransmission) of a message M with a positive acknowledgment of message N adds an edge from graph node M to graph node N
- Transmission (or retransmission) of a negative acknowledgment of message N deletes the graph node N and all its in and out edges.

Lemma 1. The graph G_A is acyclic.

Proof. In the construction of G_A, an edge is added from node M to node N if message M acknowledges message N; thus message M must have been sent after message N.

Lemma 2. If there are no unacknowledged messages, there exists a single root in the graph G_A.

Proof. If there are no unacknowledged messages, every node of G_A, except for the one most recently inserted, has an in edge. Thus, the most recently inserted node is the root of G_A.

Lemma 3. If an acyclic graph G has a single root R, every node of G is reachable from R.

Proof. This is a standard result in graph theory whose proof follows by structural induction on the set of subgraphs of G with the subgraph relation.

Lemma 4. If there are no unacknowledged messages and if the most recently transmitted message A does not contain a negative acknowledgment of Z, then the network node originating A has received Z or a previous version thereof correctly.

Proof. Consider the graph G_A constructed above. Let P be a path from A to Z in G_A. The proof is by structural induction on the set of subpaths of P that start at A with the subpath relation.

Base. $P = \langle\{A, Z\}, \{(A, Z)\}\rangle$. The lemma follows.

Step. $P \neq \langle\{A, Z\}, \{(A, Z)\}\rangle$. Assume that the lemma holds for all subpaths P' of P that start at A. Let Y be the immediate predecessor of Z on the path P, and let P' be the subpath of P from A to Y. By the inductive assumption, the network node originating A has received Y or a previous version thereof correctly. Furthermore, Y contains a positive acknowledgment of Z. Hence, A knows of the existence of Z. Since A does not contain a negative acknowledgment of Z, the network node originating A has received Z or a previous version thereof correctly.

Theorem 1. If there are no unacknowledged messages and no outstanding negative acknowledgments, then the node that sent the most recent message has received all messages correctly.

Proof. Consider the graph G_A constructed above. By Lemmas 1 and 2, G_A is acyclic and has a single root A, which corresponds to the most recent message sent. Let M be an arbitrary message. Then, by Lemma 3, there exists a path P from A to M. By Lemma 4, the network node originating A has received M or a previous version thereof correctly.

We can also provide predicates on the message history that determine whether a given node has received a specific message correctly and thus, by enumeration, whether all nodes have received a specific message correctly. Again the proof of these properties is based on the representation of messages and acknowledgments as a finite directed acyclic graph G_A. The graph differs from the one above in that edges of the graph represent positive or negative acknowledgments or retransmissions. The construction of the graph G_A is as follows:

- Transmission (retransmission) of a message M (M_1) adds a graph node M (M_1) to G_A

- Transmission of a message M with a positive (negative) acknowledgment of message N adds an edge labeled positive (negative) from graph node M to graph node N
- Retransmission of a message M_1 adds an edge labeled retransmission from graph node M_1 to graph node M.

Lemma 5. If there exists a path of positive acknowledgments in G_A from A to Z and no negative acknowledgment has been issued for any message M on the path by A or by a message N that has been acknowledged (directly or indirectly) by A, then the network node originating A has received Z correctly.

Proof. Consider the graph G_A constructed above. Let P be a path from A to Z in G_A. The proof is by structural induction on the set of subpaths of P that start at A with the subpath relation.

Base. $P = \langle\{A, Z\}, \{(A, Z)\}\rangle$. The lemma follows even without the second hypothesis.

Step. $P \neq \langle\{A, Z\}, \{(A, Z)\}\rangle$. Assume that the lemma holds for all subpaths P' of P that start at A. Let Y be the immediate predecessor of Z on the path P, and let P' be the subpath of P from A to Y. By the inductive assumption, the network node originating A has received Y correctly.

Suppose now that the network node originating A did not receive Z correctly. Then, since the network node originating A saw Y's positive acknowledgment for Z, either A contains a negative acknowledgment for Z or there exists a negative acknowledgment for Z contained in a message that the network node originating A has acknowledged. In either case, we have a contradiction of the second hypothesis.

Theorem 2. If there exists a path of positive acknowledgments or retransmissions in G_A from A to Z and no negative acknowledgment has been issued for any message M on the path by A or by a message N that has been acknowledged (directly or indirectly) by A, then the network node originating A has received Z or some version thereof correctly.

Proof. By direct extension to the proof of Lemma 5.

The various situations involved in Lemma 5 and Theorem 2 are depicted in Figure 3.1.

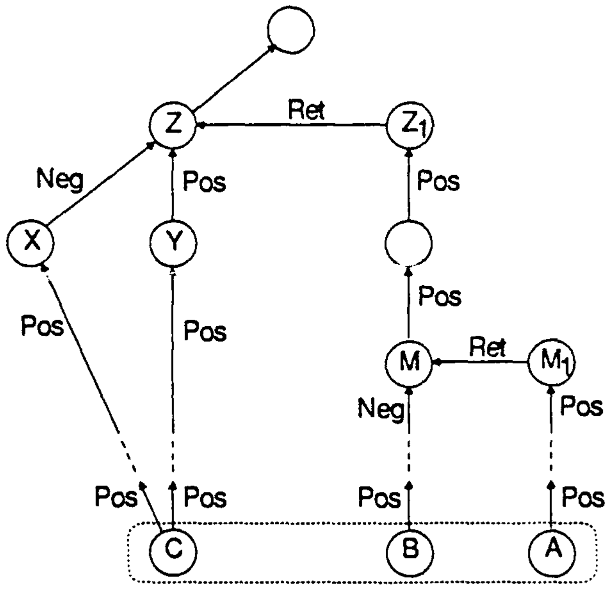


Figure 3.1: Determination of the Receipt of a Message. Analysis of the graph enables one to conclude that message Z has been received by the node broadcasting the three messages A, B and C.

We are currently working on a more formal statement of the protocol and an accompanying more formal proof of this reliability property.

3.7 Performance Model

In order to compare the broadcast protocol with existing link-level protocols, a simple queuing theory analysis has been done. To ensure a fair comparison, we require for the reliable broadcast protocol that every node broadcast a message, possibly null, within a prescribed time interval to ensure that the originators of broadcast messages can be certain that every recipient has the message. We shall compare the time to obtain such positive acknowledgment with the corresponding time for other protocols. This positive-acknowledgment comparison imposes a heavy burden on the negative-acknowledgment broadcast protocol, but by almost any other measure the broadcast protocol is so much better that there is little point in even making a comparison.

Consider first a simple point-to-point positive-acknowledgment system.

Let the

Number of nodes in the network be n
Time to transmit a message be s
Ratio of the time to transmit a message to the time to transmit an acknowledgment be p
Proportion of messages awaiting broadcast be r
Rate of demand for message transmission be ν.

Then the load on the broadcast medium is

$$\lambda = s\nu(1 - r + nr)(1 + p),$$

and the time to broadcast a message and receive the corresponding acknowledgments is

$$\frac{s(1+p)(1-r+nr)}{1-\lambda}.$$

This may be rather optimistic, since it assumes random initiation of broadcasts and thus understates the amount of contention that arises between message broadcasts and attempts to acknowledge prior broadcasts. Careful implementation of such a protocol may succeed in reducing such contention. To some extent, it also disregards the effects of disparities in the lengths of messages and acknowledgments.

Turning to the reliable broadcast protocol, we must define the time period for which a node must wait before sending a null message to indicate that it is still present in the network and has received prior broadcasts. We also denote by q the probability that a node will not have transmitted a regular message within d and thus will require a null message.

Then, the load on the broadcast medium is

$$\lambda = s\nu(1-r)(1+p) + s\nu r(1+npq)$$

where

$$q = \exp\left(-\frac{\nu}{n}\left(\frac{s\lambda}{1-\lambda} + d\right)\right),$$

while the delay incurred before it is certain that the broadcast message has been received by all destinations is

$$\frac{s(1-r)(1+p) + rs(1+npq)}{1}.$$

These equations were solved numerically by using a simple Pascal program to obtain the results shown in the following figures.

Figure 3.2 compares the time to receipt of a positive acknowledgment in systems of three sizes–10, 20 and 50 nodes. We assume that transmission of a typical message requires use of the broadcast medium for 1 ms, while an acknowledgment alone requires only 0.1 ms. In this figure, we assume that a node will transmit an acknowledgment within 100 ms if it has not sent any other message within that time. As expected, the results of the analysis show that a 10-node point-to-point protocol becomes saturated at about 90 messages per second, a 20-node system at about 45 messages per second, and a 50-node system at about 18 messages per second. In contrast, the 10- and 20-node broadcast protocol results are almost identical, becoming saturated at about 1000 messages per second; at that load all acknowledgments can be piggybacked onto other messages. In the 50-node system, the broadcast protocol becomes saturated at about 250 messages per second. At all sizes, the broadcast protocol provides an order-of-magnitude increase in potential traffic load.

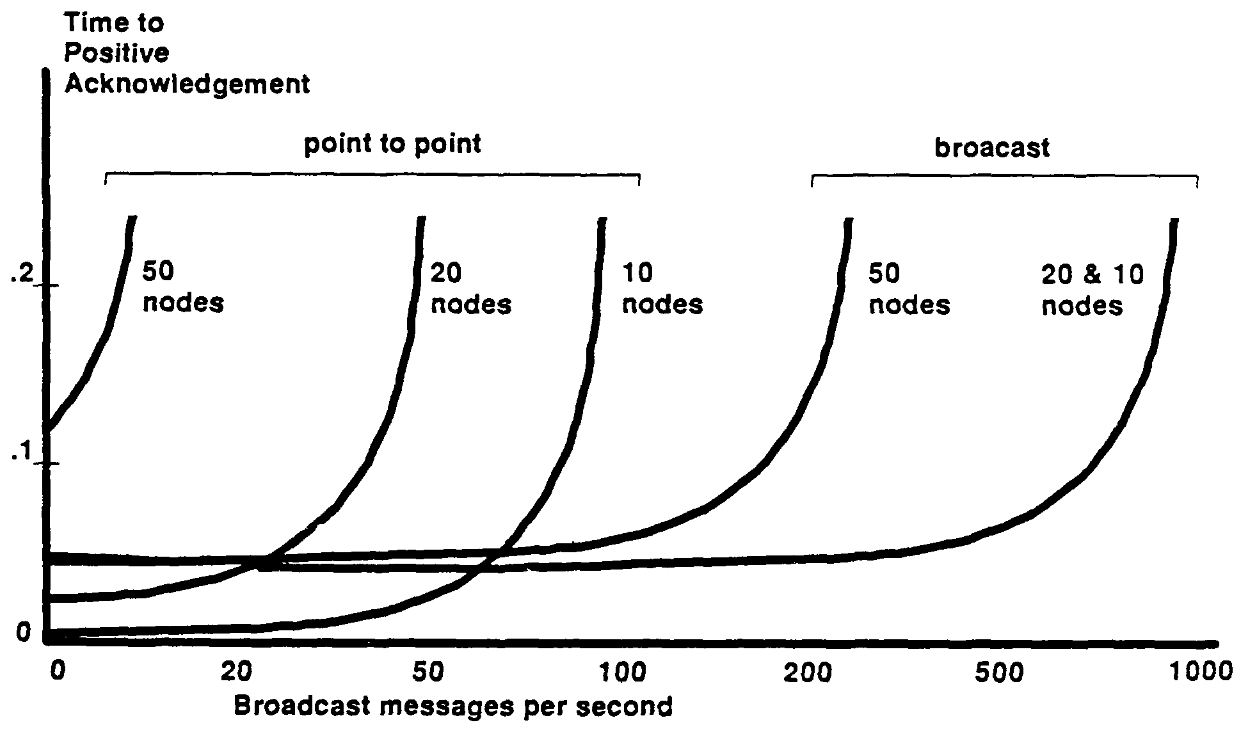


Figure 3.2: Comparison of Times to Positive Acknowledgment for Point-to-point and Broadcast Protocols.

It is also appropriate to investigate the effect of a node's waiting time before broadcasting an acknowledgment when it has no other message to transmit. When the delay is as long as one second, Figure 3.3 shows that the results for systems containing 10, 20 and 50 nodes are almost identical and that all become saturated at about 1000 messages per second. But, of course, the time to positive acknowledgment is long. Reducing the delay to 10 ms greatly reduces the time to positive acknowledgment, but now causes all sizes to become saturated below 1000 messages per second. In each case, however, the broadcast protocol is able to support much more traffic than a point-to-point protocol, with comparable times to positive acknowledgment.

Finally, we consider the possibility that not all of the messages require broadcasting to all other nodes; some messages are intended for only a single destination. Figure 3.4 shows results for 100%, 10%, and 1% of all messages requiring broadcast. The point-to-point protocol shows substantially better performance as the proportion of broadcast messages diminishes. The broadcast protocol results are identical except for the 50 node, 100% broadcast case. Clearly, the advantage of the broadcast protocol lessens commensurately as the proportion of broadcast messages is reduced.

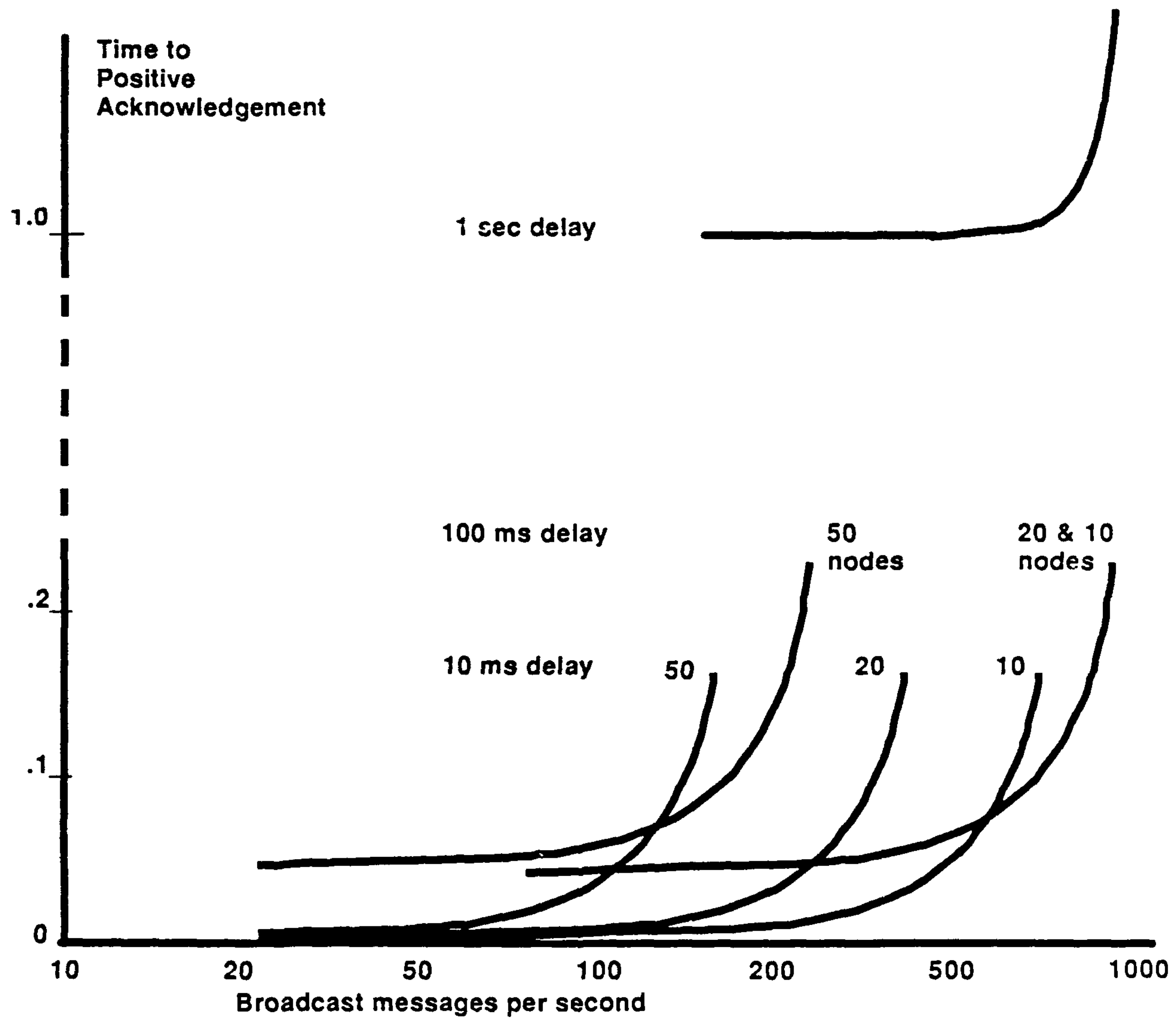


Figure 3.3: The Effect of Delay Time on the Protocol Performance.

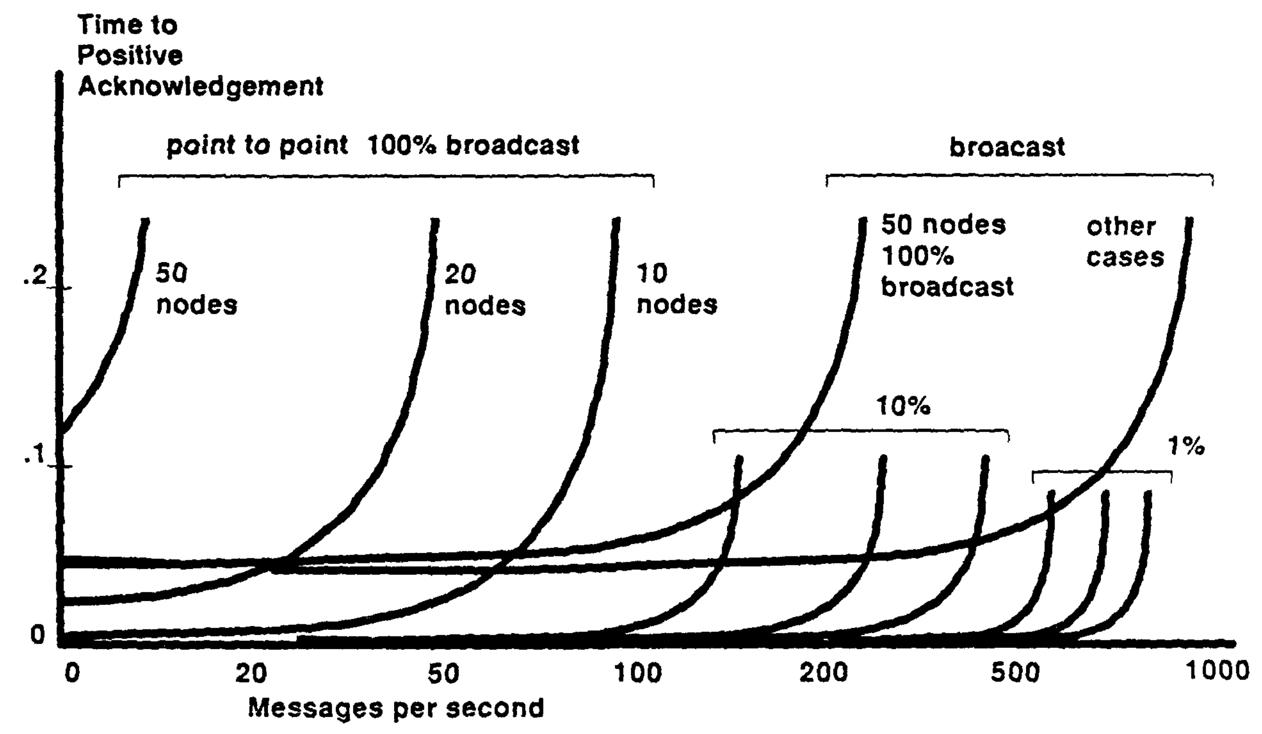


Figure 3.4: The Effect of the Proportion of Messages Broadcast on the Protocol Performance.

3.8 Broadcast Algorithms for Mutual Exclusion and Distributed Update

We have started to consider various applications for which the reliable broadcast protocol would be advantageous. Mutual exclusion, locking, and synchronization algorithms exemplify an application in which broadcast communication can provide substantial benefits. A simple mutual exclusion protocol, based on the algorithms of Ricart and Agrawala[8], uses claim, reject, and release messages. A node seeking the lock broadcasts a claim message and waits. If no other node disputes this claim by broadcasting a reject message within that period, the node may enter the critical section. A node may broadcast a release message as it leaves the critical section, though such messages are necessary only when other nodes are waiting. Contention among nodes can be resolved by timestamps in the usual way[9].

The above protocol will work under ideal conditions, but is hardly robust; any one of a number of errors could result in more than one node in the critical section simultaneously. There is no easy way to guarantee recovery when a node fails while in the critical section other than through an audit and restoration of the shared resource. However, much can be done to make the mutual exclusion protocol more robust.

The protocol can be refined by defining a caucus of nodes responsible for administration of the lock. Only members of the caucus maintain a record of lockholders and thus need to respond to claim messages, rejecting them because of conflict or because fewer than a majority of those in the caucus are currently able to communicate. While this ensures reliability in the presence of network partitioning, or failure of caucus members, it does not, of course, guarantee against failure of the node holding the lock.

A similar protocol permits updates and commitment in a replicated database. The caucus is composed of the set of nodes holding copies of the data in question. Updates are performed by a single broadcast message conveying the update request and whatever additional timestamps are needed by the conflict detection algorithms, which unfortunately are not themselves simplified by the broadcasting. After a delay during which the update can be rejected by reason of conflict or lack of a majority, it is automatically committed. The protocol also provides for reading data reliably from the database, for readmitting a failed node (particularly a caucus member) and for rejoining a partitioned network.

3.9 Conclusions

Aside from intellectual interest, the utility of such protocols depends on the cost and speed ratios for processing and communication, the load on the communication medium, the nature of the traffic, and the effect of the delays required by the broadcast protocols.

Part IV

Extending Interval Logic to Real Time Systems

4.1 Abstract

Interval logic is a temporal logic that provides a higher-level framework for specifying distributed systems. The concepts of intervals and interval composition form the basic structure of many specifications. Interval logic allows such conceptual requirements to be stated rather directly and intuitively.

Temporal logic has suffered from its orientation towards eventuality rather than immediacy in real time; indeed, pure temporal logic makes no reference to time! There are many real time properties that are critical to the specification of distributed systems. We have been able to extend interval logic to allow real time bounds on intervals and to allow events to be defined by real time offsets from other events. The extension is clean and sufficient to describe real time constraints directly and easily.

The interval logic is demonstrated by application to the lift specification example.

4.2 Introduction

Temporal logic has been found useful for specifying distributed asynchronous systems. Traditionally, such specifications have been expressed as interacting state machines, but that approach inevitably suffers from over specification for the state machines represent an implementation. If the application is such that only one implementation is envisaged, an implementation oriented specification may be acceptable; but other applications, for example communications protocol specifications, envisage many distinct implementations. By specifying the minimum required externally visible behavior, leaving all other aspects to lower levels of description, one can be obtain a more general specification that reflects the necessary requirements of the distributed system or protocol. A specification that is oriented towards one implementation may discourage or even preclude other equally valid implementations. Specifications expressed in temporal logic do not suffer as severely from implementation bias as do state machine specifications.

A specification for a distributed system can serve to define the externally observable function of the system, in effect the service provided by the system. Such specifications are called service specifications. A service specification regards the entire distributed system as a single entity, with multiple interfaces at separate nodes of the distributed system. The specification defines how operations at each interface, performed asynchronously, affect results at other interfaces. Ideally, a service specification defines only the behavior visible at the external interfaces, without suggesting any internal structure for the system.

Many service specifications define that all operations at the external interfaces be serializable, a characteristic that is often desirable for user interfaces. Such specifications can often be expressed with simpler specification languages that provide only the concepts of parallel operation and of atomicity.

Alternatively, a specification can define the manner in which the separate components of the distributed system interact with each other so as to provide the required function. Such a specification is called an implementation specification or a protocol specification. An implementation specification defines separately the behavior of each component, so that each distributed component can be implemented separately. The specifications describe how the components communicate with each other using a communication facility, which is defined by a service specification, as is shown in Figure 4.1. The communication facility is, of course, itself a distributed system for which there is, in addition to the service specification, also an implementation specification dependent on an even more primitive communication mechanism. In many distributed systems, the hierarchy of such specifications is several levels deep.

If there are to be several independent implementations of some of the components, in the future even if not immediately, it is important that the implementation specification describe only how the components interact with each other without unnecessarily constraining the internal implementation of any component. The ideal specification is one in which

- any component, that satisfies its specification, will operate satisfactorily in the system, and
- any component, that operates satisfactorily in the system, will satisfy the specification.

If both a service specification and an implementation specification have been constructed for a distributed system, it is possible to validate the implementation specification by confirming that it satisfies the service specification. This ability is very valuable for the implementation specification is often quite complex and prone to error, while the service specification is much shorter and simpler. Unfortunately, the current state of the art, and particularly of tools, has not yet advanced to the point at which such a validation is feasible for typical distributed systems.

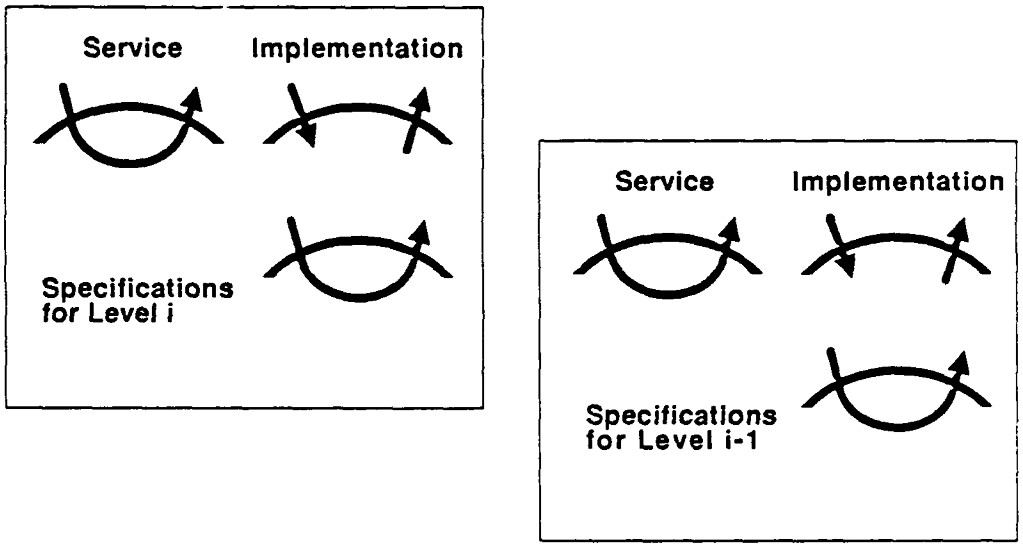


Figure 4.1: Specification of a Level in the Protocol Hierarchy.

4.3 The Basic Interval Logic

In a previous survey paper[10], we examined how several different temporal logic approaches express the conceptual requirements for a simple protocol. Our conclusions were both disappointing and encouraging. On one hand, we saw how the very abstract temporal requirements provided an elegant statement of the minimal behavior for an implementation to conform to the specification. We were able to distill a set of requirements expressing the essence of the desired behavior, stating only requirements without implementation-constraining expedients.

While we were happy with the level of conceptualization of the specifications, their expression in temporal logic was rather complex and difficult to understand. The relatively low level of the linear-time temporal logic operators encourages the inclusion of additional state components that are not properly part of the specification, but that help to establish the context necessary to express the requirements. Without these components, context can only be achieved by complex nestings of temporal until constructs to establish a sequence of prior states. The survey paper showed how the introduction of state simplifies the temporal logic formulas at the expense of increasing the amount of "mechanism" in the specification. The specification that defined only the minimum required externally visible behavior, without any additional internal state components, was also the least readable. As a result of this survey, the interval logic was developed to allow the specification of distributed systems in a manner that corresponds more closely to the intuitive intent and understanding of the designers.

At the heart of our interval logic are formulas of the form:

$$[\, I \,]\alpha$$

Informally, the meaning of this is: "The *next* time the interval I can be constructed, the formula α will 'hold' for that interval." This interval formula is evaluated within the current interval context and is vacuously satisfied if the interval I cannot be found. A formula 'holds' for an interval if it is satisfied by the interval sequence, with the present state being the beginning of the interval.

The unary $\Box$ and $\Diamond$ temporal logic operators retain their intuitive meaning within interval logic. The formula $[\,I\,]\Box\,\alpha$ requires that property α must hold throughout the interval, while $[\,I\,]\Diamond\alpha$ expresses the property that sometime during the interval I, α must hold. For simple state predicate P, the interval formula $[\,I\,]P$ expresses the requirement that P be true in the first state of the interval.

Interval formulas compose with the other temporal operators to derive higher-level properties of intervals. The formula

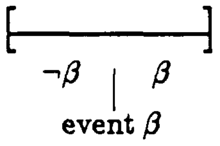
$$[\,I\,][\,J\,]\alpha$$

states that the first J interval contained in the next I interval, if found, will have property α. The property that all J intervals within interval I have property α would be expressed as $[\,I\,]\Box[\,J\,]\alpha$. More globally, the formula $\Box[\,I\,]\alpha$ requires all further I intervals to have property α.

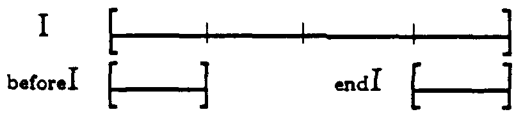
Each interval formula $[\,I\,]\,\alpha$ constrains α to hold only if the interval I can be found. Thus only when the context can be established need the interval property hold. To *require* that the interval occur, one could write $\neg\,[\,I\,]$False. The interval language defines the formula $*\,I$ to mean exactly this.

Thus far, we have described how to compose properties of intervals without discussing how intervals are formed. At the heart of a very general mechanism for defining and combining intervals is the notion of an *event.* An event, defined by an interval formula β, occurs when β changes from False to True, i.e., when it *becomes* true. In the simplest case, β is a predicate on the state, such as $x > 5$ or at Dq . Note that, if the predicate is true in the initial state, the event occurs when it changes from False to True, and thus only after the predicate has become False.

Intervals are defined by a simple or composed interval term. The primitive interval, from which all intervals are derived, is the *event interval.* An event, defined by β, denotes the *interval of change* of length 2 containing the $\neg\beta$ and β states comprising the change. Pictorially, this is represented as



Two functions, before and end, operate on intervals to extract unit intervals. For interval term I, before$I$ denotes the unit interval containing the first state of interval I. Similarly, end$I$ denotes the unit interval at the end. Application of the end function is undefined for infinite intervals. Again, pictorially, the intervals selected are



For a P predicate event, the following formulas are valid.

$[\,\text{end}\,P\,]P$

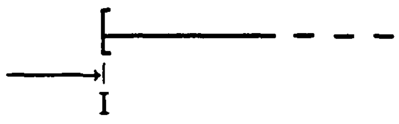
$[\,\text{before}\,P\,]\neg P$

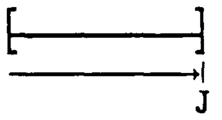
$[\,P\,]\neg P$

4.3.1 The Interval Operators $\Rightarrow$ and $\Leftarrow$

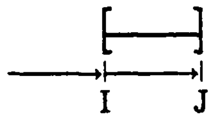
Two generic operators exist to derive intervals from interval arguments. We take the liberty of overloading these operators to allow zero, one or two interval-value arguments. Intuitively, the direction of the operator indicates in which direction and in which order the interval endpoints are located. The endpoint at the tail of the arrow is first located, followed by a search in the direction of the arrow for the second endpoint. A missing parameter causes the related endpoint to be that of the outer context.

The interval term $I \Rightarrow$ denotes the interval commencing at the end of the next interval I and extending for the remainder of the outer context. The right arrow operator, in effect, locates the *first* I interval, relative to the outer context, and forms the interval from the *end* of that I interval onward. With only a second argument present, $\Rightarrow J$ denotes the interval commencing with the first state of the outer context and extending to the *end* of the *first* J interval. Thus,



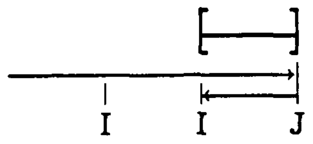


The term $I \Rightarrow J$, with two interval arguments, represents the composition of the two definitions. This constructs the interval starting at the end of interval I and extending to the end of the *next* interval J located in the interval $I \Rightarrow$. Given this definition, the interval formula $[\ I \Rightarrow J\ ]\alpha$ is equivalent to $[\ I \Rightarrow\ ][\ \Rightarrow J\ ]\ \alpha$. Recall that the formula $[\ I \Rightarrow J\ ]\ \alpha$ is vacuously true if the $I \Rightarrow J$ interval cannot be found. Pictorially, the interval selected is



The right arrow operator with no interval arguments selects the entire outer context.

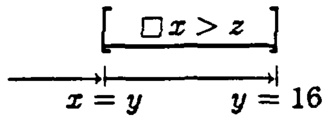
The left arrow operator $\Leftarrow$ is defined analogously. For interval term $I \Leftarrow J$, the first J interval in context is located. From the end of this J interval, the *most recent* I interval is located. The derived interval $I \Leftarrow J$ begins with $\text{end}I$ and ends with $\text{end}J$. Thus,



Similarly, the interval term $I \Leftarrow$ selects the interval beginning with the end of the last I interval and extending for the remainder of the context. For a context in which an interval I occurs an infinite number of times, the formula $[\ I \Leftarrow\ ]\,\alpha$ is vacuously true. The interval terms $\Leftarrow$ and $\Leftarrow J$ are strictly equivalent to $\Rightarrow$ and $\Rightarrow J$, respectively.

The following examples illustrate the use of the interval operators.

$$[\,x = y \;\Rightarrow\; y = 16\,]\,\Box\; x > z \tag{4.1}$$



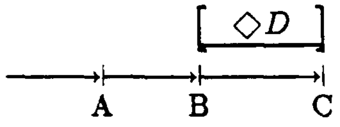
For the interval beginning with the next event of the variable x becoming equal to y and ending with y changing to the value 16, the value of x is asserted to remain greater than z. The first state of the interval is thus the state in which x is equal to y and the last state is that in which y is next equal to 16. Note that the events $x = y$ and $y = 16$ denote the next *changes* from $x \neq y$ and $y \neq 16$.

To modify the above requirement to allow $x > z$ to become False as y becomes 16, one could write

$$[\,x = y \;\Rightarrow\; \text{before}(y = 16)]\,\Box\; x > z \tag{4.2}$$

Nesting interval terms provides a method of expressing more comprehensive context requirements. Consider the formula

$$[\,(A \Rightarrow B) \Rightarrow C\,]\Diamond D \tag{4.3}$$



The formula requires that, if an A event is found, the subsequent B to C interval, if found, must sometime satisfy property D. The outer $\Rightarrow$ operator selects the interval commencing at the end of its first argument, in this case, at the end of the selected $A \Rightarrow B$ interval. The interval then extends until the next C event – establishing the necessary context.

In the previous example, the formula was vacuously true if any of the events A,B, or C could not be found in the established context. In order to easily express a requirement that a particular event or interval *must* be found if the necessary context is established, we introduce an interval term modifier $*$. For interval term I, $*I$ adds an additional requirement that B must be found in the designated context. The formula

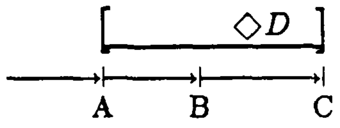
$$[\,(A \Rightarrow\ *B) \Rightarrow C\,]\Diamond D \tag{4.4}$$

strengthens formula (3) by adding the requirement that, if an A event occurs, a subsequent B event *must* occur. This is equivalent to formula (3) conjoined with $[\,A \Rightarrow\,]*B$.

The $*$ modifier can be applied to an arbitrary interval term. The formula $[\ *(A \Rightarrow B) \Rightarrow C\ ]\Diamond D$, for example, would be equivalent to (3) conjoined with $*(A{\Rightarrow}B)$, or equivalently, $*A \ \ \wedge\ \ [\ A\ \Rightarrow\ ]\ *B$. The $*$ modifier adds only linguistic expressive power and can be eliminated by a simple reduction (given in the Appendix).

As an example of specifying context for the end of the interval, consider the formula

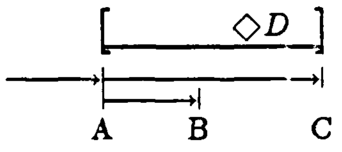
$$[\,A \Rightarrow (B \Rightarrow C)\,]\Diamond D \tag{4.5}$$



Here, the interval begins with the next occurrence of A and terminates with the first C that follows the next B.

By modifying formula (3) to begin the interval at the beginning of $A \Rightarrow B$, i.e.,

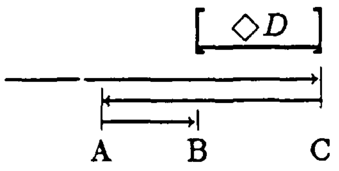
$$[\,\text{before}(A \Rightarrow B) \Rightarrow C\,]\Diamond D \tag{4.6}$$



we obtain a requirement similar to that of (5), but allowing events B and C to be ***arbitrarily ordered.***

Introducing the use of backward context, to find the interval $A \Rightarrow B$ in the context of C, we have

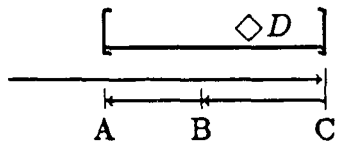
$$[\,(A \Rightarrow B) \Leftarrow C\,]\Diamond D \tag{4.7}$$



Here the occurrence of the first C event places an endpoint on the context, within which the most recent $A \Rightarrow B$ interval is found. Note the order of search: looking forward, the next C is found, then backward for the most recent A, then forward for the next B. Thus, the formula is vacuously true if no B is found between C and the most recent A.

As a last example, consider

$$[\,\text{before}(A \Leftarrow B) \Leftarrow C\,]\Diamond D \tag{4.8}$$



The interval extends back from the first C event to the beginning of the most recent $A \Leftarrow B$ interval.

4.3.2 Parameterized Operations

Within the language of our interval logic we include the concept of an *abstract operation*. For an abstract operation O, state predicates $\mathrm{at}O$, $\mathrm{in}O$, and $\mathrm{after}O$ are defined. These predicates carry the intuitive meanings of being "at the beginning", "within", and "immediately after" the operation. Formally, we use the following temporal axiomatization of these state predicates.

1. $[\, \mathrm{at}O \Rightarrow \mathrm{before}\ \mathrm{after}O \,]\,\Box\,\mathrm{in}O$
2. $[\, \mathrm{after}O \Rightarrow \mathrm{before}\ \mathrm{at}O \,]\,\Box\,\neg\mathrm{in}O$
3. $[\, \neg\mathrm{at}O \Rightarrow \mathrm{after}O \,]\,\Box\,\neg\mathrm{at}O$
4. $[\, \neg\mathrm{after}O \Rightarrow \mathrm{at}O \,]\,\Box\,\neg\mathrm{after}O$

Axioms 1 and 2 together define $\mathrm{in}O$ to be true exactly from $\mathrm{at}O$ to the state immediately preceding $\mathrm{after}O$. Axiom 3 allows $\mathrm{at}O$ to be true only at the beginning of the operation, and axiom 4 requires that $\mathrm{after}O$ be true only immediately following an operation. Note that, in axiom 1 for example, the predicate $\mathrm{at}O$ used as an event term defines the interval commencing with the *entry* to the operation.

The axioms do not imply any specific granularity, duration or mapping of the operation symbol to an implementation. *Any interpretation of these* *state predicate symbols satisfying the above axioms is allowed.* In addition, no assumption of operation termination is made. To require an operation to always terminate, one could state as an axiom

$$[\,{}_{\text{at}}O \Rightarrow \ast\,{}_{\text{after}}O\,]\text{True}$$

Abstract operations may take entry and result parameters. For an operation taking n entry parameters of types $T_1, \ldots, T_n$, and m result parameters of types $T_{n+1}, \ldots, T_{n+m}$, the at and after state predicates are overloaded to include parameter values. ${}_{\text{at}}O(v_1, \ldots, v_n)$ is true in any state in which ${}_{\text{at}}O$ is true and the values of the parameters are $v_1, \ldots, v_n$. The predicate after is similarly overloaded.

As an example of an interval requirement involving parameterized operations, consider an operation O with a single entry parameter. To require that this parameter increase monotonically over the call history, one could state

$$\forall a, b\ \Box[\,{}_{\text{at}}O(a) \Rightarrow {}_{\text{at}}O(b)\,]\ b > a$$

Since a and b are free variables, for all a and b such that we can find an interval commencing with an ${}_{\text{at}}O(a)$ and ending with an ${}_{\text{at}}O(b)$, b must be greater than a. Recall that the formula is vacuously true for any choice of a and b such that the interval cannot be found.

It is also useful to be able to designate the *next* occurrence of the operation call, and to bind the parameter values of that call. The event term ${}_{\text{at}}O : (a)$ designates the next event ${}_{\text{at}}O$ and binds the free variable a to the value of the parameter for that call. Thus the previous requirement constraining all pairs of calls, can be restated in terms of successive calls as

$$\Box[\,{}_{\text{at}}O(a) \Rightarrow {}_{\text{at}}O : (b)\,]\ b > a$$

The requirement is now that for every a, the call ${}_{\text{at}}O(a)$ is followed by a call of O whose parameter is greater than a. This parameter binding convention has a general reduction, which we omit here. For this specific formula, the reduction gives

$$\Box[\ {\scriptstyle\mathrm{at}}O(a) \Rightarrow\ ]\ \ (\ [{\scriptstyle\mathrm{end\ at}}O\ ]{\scriptstyle\mathrm{at}}O(b)\ ) \supset [\ \Rightarrow {\scriptstyle\mathrm{at}}O\ ]b > a$$

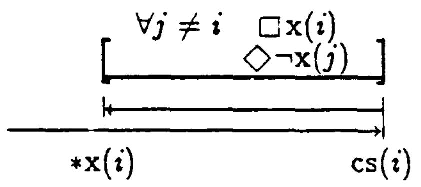
4.4 Some Example Specifications

Consider a queue with two operations, Enq which takes a single parameter value, which it enqueues, and Dq which removes the value at the front of the queue and returns that value as its result. We assume in this specification that the queue is unbounded, and require that values enqueued must be distinct. No assumptions are made about the atomicity of, or temporal relationships between, the Enq and Dq operations. These operations can overlap in an arbitrary manner. We do assume that at most one instance of the Enq and Dq operations will be active at any given time.

The specification expresses the fundamental first-in first-out behavior that characterizes a queue. It requires that, for all a and b, if we dequeue b, then any other value a will be dequeued in the interim if and only if it was enqueued prior to b. Further axioms are needed to express liveness requirements on the two operations.

$$\text{Queue.}\quad [\ \Leftarrow {\scriptstyle\mathrm{after}}\mathrm{Dq}(b)\ ](*{\scriptstyle\mathrm{after}}\mathrm{Dq}(a) \quad\equiv\quad *({\scriptstyle\mathrm{at}}\mathrm{Enq}(a) \Leftarrow {\scriptstyle\mathrm{at}}\mathrm{Enq}(b)\ ))$$

As a second example, consider a specification to ensure exclusive access to a shared critical section by some set of processes. Each process is to make an independent decision based on a shared global data structure. In stating the specification, we assume a state predicate cs(i) which, for process i, indicates that i is in the critical section. For a shared global data structure, we assume a state predicate x(i) which, for process i, indicates i's intention to enter the critical section. We wish to state minimal requirements on the use of state predicate x by a process to ensure mutual exclusion. Pictorially we represent the required behavior as follows:



As shown, an entry of the critical section by process i must be preceded by an earlier setting of $x(i)$ to true. Throughout this interval $x(i)$ must remain true, and, for every other process j, there must be some moment within the interval at which $x(j)$ is false. This specification imposes no requirement on the order or frequency of inspecting the $x(j)$s; it suffices that, *at some time* during the interval, each $x(j)$ is false. Herein lies the basic reason for exclusion. $x(i)$ remains true through the interval, and no other $x(j)$ can be true for that interval. Thus no other process j can find $x(i)$ false between the time that i signals his intention and the time that i leaves the critical section (or abandons his claim). The specification does not, however, ensure the absence of deadlock.

In interval logic, we express these requirements as follows.

Init. $\forall m \neg x(m)$

A1. $i \neq j \quad \supset \quad [\, x(i) \Leftarrow cs(i)\,] \Diamond \neg x(j)$

A2. $cs(i) \quad \supset \quad x(i)$

Given an initial condition in which all processes have relinquished their claims, axiom A1 expresses our previous pictorial requirement that, if process i enters the critical section, then for the interval back to the most recent setting of $x(i)$, each $x(j)$ must be found to be false. Axiom A2 requires that $x(i)$ remains true while i is in the critical section. We have not needed to state explicitly that there must be a setting of $x(i)$ prior to the entry; this is deducible from the specification. Similarly we can deduce that $x(i)$ remains true through that interval.

From this specification, we can demonstrate (omitted here) the mutual exclusion property that henceforth no pair of processes can both be in the critical section at the same time, i.e.,

$$\forall m \ \neg \mathrm{x}(m) \wedge i \neq j \quad \supset \quad \Box \neg(\ \mathrm{cs}(i) \wedge \mathrm{cs}(j)\ )$$

4.5 Real Time Extensions

Temporal logic has suffered from its orientation towards eventuality rather than immediacy in real time; indeed, pure temporal logic makes no reference to time! A temporal logic specification defines only invariants, eventuality, and order constraints on the sequence of states resulting from the execution of the distributed system without reference to when the states actually occur. But the specification of distributed systems typically depends criticlly on the specification of real time properties.

Surprisingly, in view of the orientation of temporal logic towards eventuality, there are useful eventuality properties, superficially independent of real time, that cannot be written without reference to real time. For example, the service specification for a lift, without consideration of the possibility of lift failure, can be expressed as a requirement that if a request is made for floor a then, eventually, the lift will be at floor a with the door open.

$$\Box\ (\ \mathrm{Request}(a) \supset \Diamond\ \mathrm{atfloor}(a) \wedge \mathrm{dooropen}(a)\ )$$

Unfortunately, any practical lift inevitably has occasions when it is out of service, expressed as

$$\Box \Diamond\ \neg \mathrm{inservice}.$$

If we are to avoid expedients such as regarding an out of service state as a terminal state, or of requiring that the lift remember the request for floor a through the out of service state (an unreasonable requirement), we would like to modify the service specification to state that the lift will eventually be at floor a unless it goes out of service first. There is no way to express that requirement; the best that can be achieved is

$$\Box \quad \Big(\text{request}(a) \supset \Diamond\, ((\text{atfloor}(a) \wedge \text{dooropen}(a)) \vee \neg\text{inservice}) \Big)$$

Careful examination shows that this specification is completely satisfied by the eventual out of service condition and it thus contributes nothing to the requirement that a request be serviced by moving to the requested floor. In effect, the lift can satisfy the specification doing nothing but wait until it breaks.

To overcome this problem, we must place a real time bound on the period of time throughout which the lift must be operational to guarantee that the service will be provided. The service specification then becomes

$$\begin{aligned}&\Box\ [\ \text{request}(a) \Rightarrow \text{request}(a)+\text{max\_service\_time}\ ]\\ &\qquad \Box\ \text{inservice} \supset \Diamond(\ \text{atfloor}(a) \wedge \text{dooropen}(a)\ )\end{aligned}$$

This states that for an interval commencing with the request and of length max_service_time, if the lift is never out of service during the interval then the service will be provided within that interval.

Thus we need to extend interval logic to include real time constraints, but we do not want, in so doing, to destroy what is valuable about the logic. Temporal logics are valuable because they allow the expression of necessary properties while precluding other forms of expression that would be inappropriate. For example, if time is represented explicitly as a numeric variable in our specifications, it is possible to express any useful temporal property, including those involving real time constraints. But, the explicit representation of time makes possible expressions that have no meaning, such as those in which a property depends on whether the time is even or odd! Thus the extension must not expose the numeric nature of time.

Further, temporal logics mask quantifications over time. An explicit representation of time could require that those temporal quantifications be explicit, complicating both the formulae and also deduction involving the formulae. If it possible to hide the quantifiers, and to process them automatically during deduction, as it is with temporal logics, we should try to do so.

The interval logic can be extended to include real time by:

- imposing real time bounds on the length of intervals, and
- allowing events to be defined by real time offsets from other events.

Defining events by real time offsets is achieved by two new operators $+, -$ syntactically defined by

$+, -$: event $\times$ duration constant $\rightarrow$ event.

Thus if E is an event then so are $E+1$ second and $E-1$ day.

Bounds on the length of intervals are provided by two relational operators, syntactically defined by

$>, <$: duration constant $\rightarrow$ boolean.

These relational operators are monadic because they relate the length of the enclosing context to the duration constant. Used within an interval, they therefore relate the length of that interval to the constant. Thus, if I is an interval, $[I]$ <1 second is a boolean predicate on the length of that interval. Similarly, we might write $[I]$ >10 seconds $\wedge \diamond$ x=4.

The relational operators can be derived form the event constructors by defining a event offset from the start of the interval and determining whether that event lies within the interval. However, the availability of the relational operators adds directness and clarity to the specifications.

These extensions to interval logic are clean and appear sufficient to describe almost all real time constraints directly and easily. They do not permit the construction of undesired expressions in which time is manipulated inappropriately.

The decidability of interval logic is unaffected by these extensions. It is not appropriate to digress here into a lengthy analysis of decidability, but rather we give only a brief outline of the necessary extensions to the decision process. A decision procedure for interval logic can be constructed as a standard semantic tableau, building a graph of possible states. The transitions between states are determined by the order of events, and thus the predicates on the states comprise the conjunction of the normal state predicates with a set of relations on the order of events.

To extend this semantic tableau decision process to the real time version of interval logic, the real time relational operators are first reduced to terms involving event constructors, as described above. The semantic tableau procedure is applied, as before, but order relations on events are regarded as linear inequalities in a real number domain, and real time event constructors are replaced by arithmetic operations in that domain. Linear arithmetic and linear inequalities in a real number domain are decidable by a Presburger procedure, thus maintaining the decidability of the logic.

4.6 The Lift Example

The objective of the Interval Logic specification is to express precisely and formally the behavior required from the lift. It is also an objective to express as few constraints on that behavior as possible while still ensuring correct behavior. It is, perhaps, easier to provide a specification that describes the lift in minute and mechanistic detail, but to do so precludes, or at least makes much less obvious, many valid implementations that are structured rather differently. Our specification, indeed, permits quite a wide range of behaviors; lifts that demonstrate some of the less obvious, but still permissible, strategies can be found in operation on occasion.

Floors

The floors are 0 to n, and the lift will not go outside this range. There can be no down button on floor 0, and no up button on floor n.

1. $\neg\text{atfloor}(-1) \bigwedge \neg\text{atfloor}(n+1)$
2. $\neg\text{light}(0, \text{down}) \bigwedge \neg\text{light}(n, \text{up})$
3. $\neg\text{request}(0, \text{down}) \bigwedge \neg\text{request}(n, \text{up})$

The lift is at only one floor at a time and moves only to adjacent floors.

4. $b \neq a \wedge \text{atfloor}(a) \quad \supset \quad \neg\text{atfloor}(b)$

5. $[\text{atfloor}(a) \Rightarrow_{\text{before}} (\text{atfloor}(a+1) \vee \text{atfloor}(a-1))]\, \Box\, \text{atfloor}(a)$

Derived Predicates

To simplify the specifications, we introduce a derived event *newrequest*, since requests are significant only if there is not already an outstanding request, if the lift does not already have its doors open at the requested floor, and if the lift is in service.

6. $\text{newrequest}(a, dir) = \text{request}(a, dir)$
$$\wedge\, \neg\text{light}(a, dir) \wedge \text{closed}(a) \wedge \text{inservice}$$

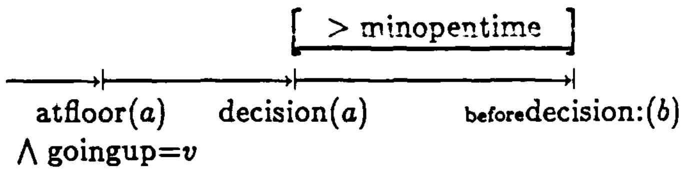
We also introduce an auxiliary event *decision* to represent the moment at which the lift decides what to do next. The event decision(a) occurs sometime after the doors open and before the lift leaves at that floor. If the lift does not stop at floor a, the event occurs some time between being at floor a and not being at floor a. Note that, at the time of the event decision(a), atfloor(a) must still be true.

7. $[\text{atfloor}(a) \Rightarrow_{\text{before}} \neg\text{atfloor}(a)] \neg * \text{open}(a) \supset * \text{decision}(a)$

8. $[(\text{atfloor}(a) \Rightarrow \text{open}(a)) \Rightarrow_{\text{before}} \neg\text{atfloor}(a)] * \text{decision}(a)$

The predicate *goingup* is introduced to represent the decision made by the lift about which direction to move. The predicate is true if the next floor that will be visited is above the current floor, and false if it is below. It must, of course retain that value until the next decision point. The curious option of remaining at the same floor and thus making a second decision at that floor is necessary in the case that the lift arrives at a floor in response to a request indicating continued travel in the same direction, but the request then made inside the lift is for travel in the other direction. The real time constraint is imposed to allow the passengers time to enter the lift and press a button.

9. $[((\text{atfloor}(a) \wedge \text{goingup} = v) \Rightarrow \text{decision}(a))$
$\Rightarrow_{\text{before}} \text{decision} : (b)] > \text{min\_open\_time}$
$$\wedge\, b > a \supset \Box\text{goingup}$$
$$\wedge\, b < a \supset \Box\neg\text{goingup}$$
$$\wedge\, b = a \supset \Box\text{goingup} = v$$

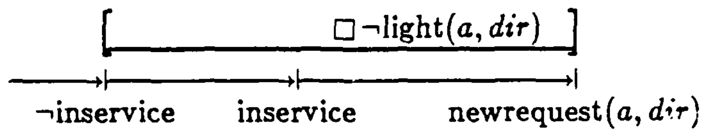


Lights

The lights are used not only to represent the lights visible to the passengers, but also to provide the memory of pending requests. Others might prefer to introduce an additional predicate to represent the pending requests explicitly.

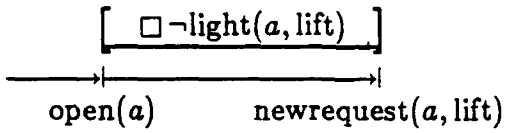
While out of service the lights must not be lit, and following a return to service the lights must not be lit until a new request has been made.

10. $[\neg\text{inservice} \Rightarrow (\text{inservice} \Rightarrow_{\text{before}} \text{newrequest}(a, dir))]\, \Box\neg\text{light}(a, dir)$

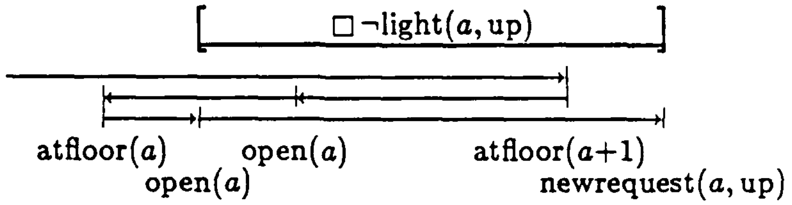


Three axioms defining when the lights must not be lit between the satisfaction of a request and the making of the next request. The case for the lift light is simple, but the other cases must consider the direction of motion of the lift and also ensure that the prohibition applies from the first time that the doors open at that floor.

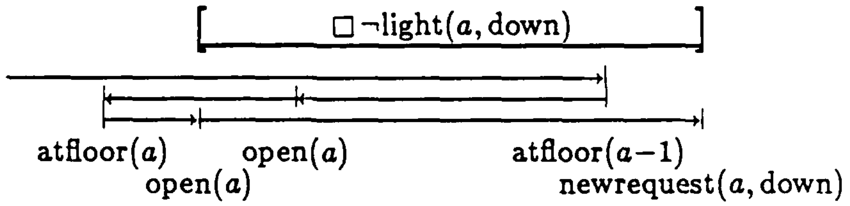
11. $[\text{open}(a) \Rightarrow_{\text{before}} \text{newrequest}(a, \text{lift})] \, \Box \neg \text{light}(a, \text{lift})$



12. $[{}_{\text{before}}((\text{atfloor}(a) \Rightarrow \text{open}(a)) \Leftarrow \text{open}(a) \Leftarrow \text{atfloor}(a+1))$
$\Rightarrow_{\text{before}} \text{newrequest}(a, \text{up})] \, \Box \neg \text{light}(a, \text{up})$

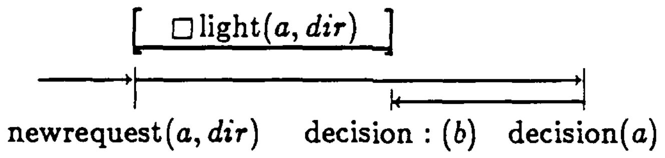


13. $[{}_{\text{before}}((\text{atfloor}(a) \Rightarrow \text{open}(a)) \Leftarrow \text{open}(a) \Leftarrow \text{atfloor}(a-1))$
$\Rightarrow_{\text{before}} \text{newrequest}(a, \text{down})] \, \Box \neg \text{light}(a, \text{down})$



An axiom that defines when the lights are required to be illuminated. the lights can be turned off as early as the previous decision point, i.e. shortly before reaching the requested floor. They can remain lit for longer but other axioms require that they be out at least by the time that the doors are open at the requested floor. The lights need only remain on so long as the lift is inservice.

14. $$\begin{aligned}&[\mathrm{newrequest}(a, dir) \Rightarrow \\ &\quad {}_{\mathrm{before}}(\mathrm{decision} : (b) \Leftarrow \mathrm{decision}(a) \wedge ((dir = \mathrm{up} \wedge \mathrm{goingup}) \vee \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad (dir = \mathrm{down} \wedge \neg \mathrm{goingup}) \vee \\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad dir = \mathrm{lift}))] \\ &\quad \Box\,\mathrm{inservice} \supset \Box\,\mathrm{light}(a, dir) \\ &\quad \wedge [\Rightarrow \neg \mathrm{inservice}]\, \Box\,\mathrm{light}(a, dir)\end{aligned}$$



Movement

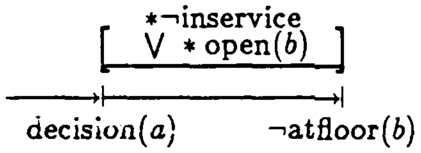
This axiom is a lift scheduling constraint that requires continued motion in one direction so long as there are further requests outstanding in that direction. When the lift decides to change its direction of motion, i.e. when goingup changes from false to true or from true to false, there must be no further request outstanding in the original direction of motion.

15. $b < a \quad \supset \quad [{}_{\mathrm{before}}\mathrm{goingup} \Rightarrow]\, \mathrm{atfloor}(a) \supset \neg \mathrm{light}(b, dir)$

16. $b > a \quad \supset \quad [{}_{\mathrm{before}}\neg \mathrm{goingup} \Rightarrow]\, \mathrm{atfloor}(a) \supset \neg \mathrm{light}(b, dir)$

When appropriate, the lift will stop and open its doors. Fast lifts need time to decelerate and stop, time that is not provided by this version of the specifications. The necessary modifications do not affect these two axioms but rather impose a speed dependent advance on the decision point defined in axiom 7.

17. $b \geq a \quad \supset \quad [(\text{decision}(a) \wedge \text{goingup} \wedge (\text{light}(b,\text{up}) \vee \text{light}(b,\text{lift})))$
$\Rightarrow \neg\text{atfloor}(b)] * \text{open}(b) \vee * \neg \mathit{inservice}$

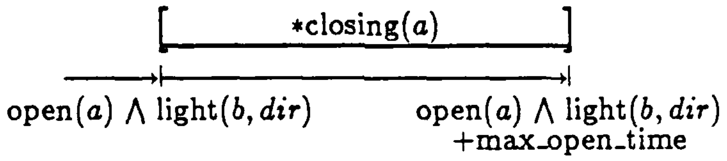


18. $b \leq a \quad \supset \quad [(\text{decision}(a) \wedge \neg\text{goingup} \wedge (\text{light}(b,\text{down}) \vee \text{light}(b,\text{lift})))$
$\Rightarrow \neg\text{atfloor}(b)] * \text{open}(b) \vee * \neg\text{inservice}$

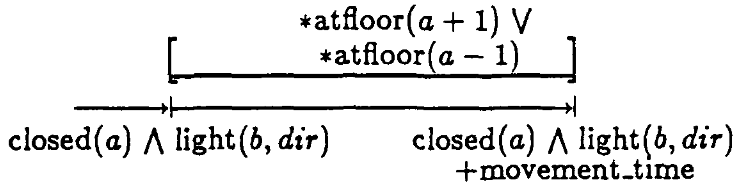
These requirements allow the wide range of behavior that we encounter in lifts, as for instance in allowing the lift to always return to the ground floor, in allowing the lift a home floor when inactive, or even in allowing the cattle car to stop at every floor regardless.

The local liveness axioms require that lift should not stay at one floor indefinately if there are requests outstanding from other floors. The first of the two axioms constrains the doors to close within a time constraint if they are not obstructed. The second requires timely movement to an adjacent floor if the lift is in service.

19. $b \neq a \quad \supset \quad [(\text{open}(a) \wedge \text{light}(b, \mathit{dir})) \Rightarrow$
$(\text{open}(a) \wedge \text{light}(b, \mathit{dir})) + \text{max\_open\_time}]$
$\Box(\text{inservice} \wedge \neg\text{obstructed}(a)) \supset * \text{closing}(a)$



$$
\begin{aligned}
20.\quad b \neq a \quad \supset \quad & [(\text{closed}(a) \wedge \text{light}(b, dir)) \Rightarrow \\
& (\text{closed}(a) \wedge \text{light}(b, dir)) + \text{movement\_time}] \\
& \Box\,\text{inservice} \quad \supset \quad (*\text{atfloor}(a+1) \vee *\,\text{atfloor}(a-1))
\end{aligned}
$$

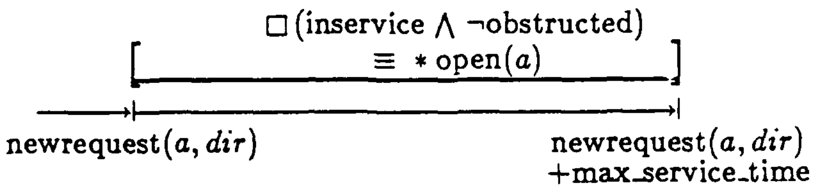


Service Specification

We must next provide our lift with a service specification. Basically, the service specification states that if a request is made for floor a, then eventually the lift will be at floor a with the doors open. As discussed above, we must temper this idealistic requirement with the possibility that the lift may go out of service. We must also allow for the possibility that the doors may be obstructed to prevent them from closing. We can now state an informal service requirement:

If a request is made for floor a by pressing a button inside the lift or at that floor, and if, throughout a sufficiently long interval commencing with the request, the lift is never out of service and the doors are never obstructed, the lift will eventually be at floor a with its doors open.

$$
\begin{aligned}
21.\quad & [\text{newrequest}(a, dir) \Rightarrow \text{newrequest}(a, dir) + \text{max\_service\_time}] \\
& \Box\,(\text{inservice} \wedge \neg\text{obstructed}) \quad \equiv \quad *\,\text{open}(a)
\end{aligned}
$$



It is possible to elaborate this requirement to allow occasional obstruction of the doors while still guaranteeing service, but at the cost of greatly complicating the specification. The complexity arises not from any inability of the specification language but from the inherent complexity of determining to what extent it is possible to obstruct the doors while still requiring the lift to provide timely service.

Door opening and closing

We now encounter a sequence of relatively simple axioms that closely control the opening and closing of the doors. Their interest lies largely in the extent to which real time constraints are necessary to specify this aspect of the lift.

Opening, open, closing, and closed are complete and mutually exclusive.

22. $\text{opening}(a) \vee \text{open}(a) \vee \text{closing}(a) \vee \text{closed}(a)$
$\wedge \quad (\text{opening}(a) \vee \text{open}(a)) \equiv \neg(\text{closing}(a) \vee \text{closed}(a))$
$\wedge \quad (\text{opening}(a) \vee \text{closing}(a)) \equiv \neg(\text{open}(a) \vee \text{closed}(a))$

23. $[\text{open}(a) \Rightarrow_{\text{before}} \text{closing}(a)]\,\Box\,\text{open}(a)$
$\wedge \quad [\text{closed}(a) \Rightarrow_{\text{before}} \text{opening}(a)]\,\Box\,\text{closed}(a)$

The lift must be at a floor to open its doors and the doors of the lift and that floor open and close together.

24. $\text{opening(lift)} \quad \equiv \quad \exists a : 0 \leq a \leq n \wedge \text{opening}(a)$

25. $0 \leq a \leq n \supset [\text{opening}(a) \Rightarrow \text{closed}(a)]\,\Box\,\text{atfloor}(a)$
$\wedge \quad \text{opening(lift)} \equiv \text{opening}(a)$
$\wedge \quad \text{open(lift)} \equiv \text{open}(a)$
$\wedge \quad \text{closing(lift)} \equiv \text{closing}(a)$
$\wedge \quad \text{closed(lift)} \equiv \text{closed}(a)$

The next five axioms place real time constraints on the sequence of opening and closing actions of the doors, allowing for the possibility that the doors may be obstructed. The last axiom states that the doors are only obstructed while closing.

26. $[\text{opening}(a) \Rightarrow \text{open}(a)]\,\Box\,\text{inservice} \supset \ < \text{opening\_time}$

27. $[\text{closing}(a) \Rightarrow \text{closed}(a)] \Box (\textit{inservice} \wedge \neg \text{obstructed}(a))$
$\supset \; < \text{closing\_time}$

28. $[\text{obstructed}(a) \Rightarrow \text{opening}(a)] \Box \text{inservice} \supset \; < \text{reaction\_time}$

29. $[(\text{obstructed}(a) \Rightarrow \text{open}(a)) \Rightarrow \text{closing}(a)] \Box \text{inservice} \supset \; < \text{dwell\_time}$

30. $[\text{open}(a) \Rightarrow \text{closing}(a)] > \text{min\_open\_time}$

31. $[\text{obstructed}(a) \Rightarrow] \text{closing}(a)$

4.7 Analysis and Conclusions

We have presented an outline of our interval logic with an extension to permit the specification of real time constraints, and have applied it to the lift specification example. We are reasonably satisfied with its success, although we feel that further experimentation is necessary. Current work is proceeding on providing the interval logic with the abilitv to describe multiprocess systems and to compose the specifications of single processes into a multiprocess specification. We are also working on integrating the interval logic into the specification language for a full verification system, and on verification techniques for concurrent programs. Future projects may investigate a human interface based on the graphical representation for interval logic rather than on the linear syntax.

We are reasonably satisfied with the style of expression of the interval logic. It appears to correspond quite closely to the intuitive forms of reasoning and explanation used by human designers while considering concurrent systems. In particular, the graphical representation for interval logic appears to be very close to typical human design sketches. The behavioral style of specification and the basing of interval formation on events derived from state changes, motivated by our observation that establishing context almost always required seeing a change in state, have been justified in our experience of the use of the logic for examples. But, despite the relatively behavioral style of specification, the specifications can be quite abstract with relatively little auxiliary state information introduced to establish context. This allows specifications expressed in interval logic to remain more general, and to impose less implementation bias, than more state oriented methods.

The real time extensions to interval logic are important for making the logic useful for the specification of real systems. Despite the power of the extension, we believe that the integrity of the logic has been maintained. That the logic with the real time extensions is still decidable is helpful in retaining the opportunity to provide mechanical support. Unaided human reasoning about concurrent systems is very falible.

The specification of, and reasoning about, complex concurrent systems is difficult, and interval logic does not eliminate that difficulty. The difficulty is inherent in the multiplicity of possible cases that must be considered, and in determining the relationships that are significant to the operation of the system. Our objective with interval logic can only be to allow the designer to express his intentions and understanding in a manner that is close to his natural intuition.

Part V

Consistency of Replicated Information in Multichannel Fault Tolerant Systems

5.1 Abstract

The need for reliable computation has induced many designs for fault tolerant computer systems based on the replication of the processors and appropriate error detection and masking algorithms. Typical of such systems are SIFT and FTMP, which use majority voting for error masking, and Stratus, which uses a dual-dual structure for error masking. It is clear that these approaches, coupled with the steadily improving reliability of components, now allow the construction of very reliable systems.

All fault tolerant systems depend on some form of error masking algorithm, coupled with error detection to allow the repair of faults. Some such systems depend on backward error correction, in which a result is computed, the acceptability of that result is checked, and in the event of error the computation of the result is repeated. Typical of such systems are classical Checkpoint-Restart systems and Recovery Blocks2. Backward error correcting algorithms necessarily incur a significant overhead for repeating the computation when an error is detected, and also involve an acceptance test on the results, a test that is usually system and application specific. We do not consider backward error correcting systems in this paper but rather we examine Forward Error Correcting systems, in which the results are computed in a redundant form that allows error masking without repeating any computation.

Two forward error correcting algorithms are currently used for masking processor errors in reliable systems, majority voting and dual-dual. The majority voting approach can mask errors caused by one faulty channel out of three, while a dual-dual approach masks one faulty channel out of four. Both approaches have the advantage that they are completely application independent. However majority voting and dual-dual both depend for their operation on exact match comparison between results of computations. Thus, for successful masking of errors, it is essential that the fault free channels should generate identical results. Both algorithms guarantee, with only a single faulty channel and with fault free channels producing identical results, that fault free channels remain error free and continue to generate identical results.

Two questions arise from this. The first concerns whether there are any single point faults that could cause fault free channels to generate different results, thus invalidating the presumptions of both majority voting and dual-dual. We describe below a class of such faults and give algorithms for precluding them. The second question relates to the possible increase in the risk of common mode faults resulting from the need for all channels to perform exactly the same computation on identical data at approximately the same time. We show below that error masking algorithms can be devised that allow each channel to perform a different computation on different data at different times.

5.2 Loss of Consistency

Figure 1 shows a majority voted three channel system, with one faulty and two working channels. The successive levels of the diagram might represent distinct units within the channel, but equally they can represent successive iterations of a computation performed by the same units. It is clear that, provided that the two working channels generate identical results initially, each voting operation will receive as inputs two identical values and one erroneous value. The voters in the two working channels will therefore both produce the same value for the majority. Thus the working channels continue to generate identical results, and consistency between working channels is maintained. However, if at any time the three channels generate different results, the voters can find no majority and the system fails.

Consider Figure 2, which shows a system of three working channels and an input to that system from a single faulty source. The nature of the fault is that the source distributes different values to each of the three channels (the values A, B, and C). Even on a broadcast bus, such faults can result from marginal timing faults or from a marginal transmitter at the source and receivers with slightly different, but within specification, characteristics. More complex communication mechanisms, particularly where software is involved, permit many more such faults. The figure shows that, if the faulty source distributes different values to each channel, the three channels generate different results, the voters can find no majority.

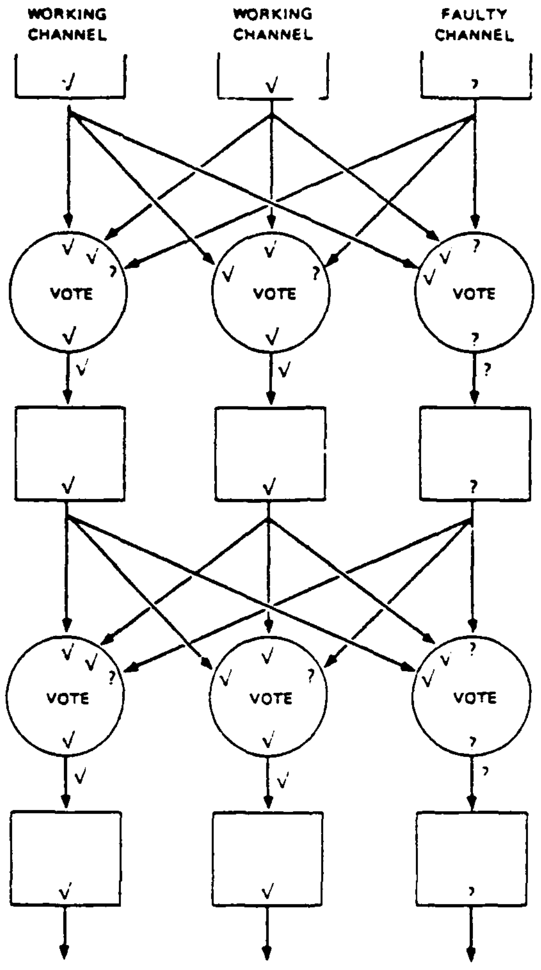


Figure 5-1: A Three-Channel Majority Voted System

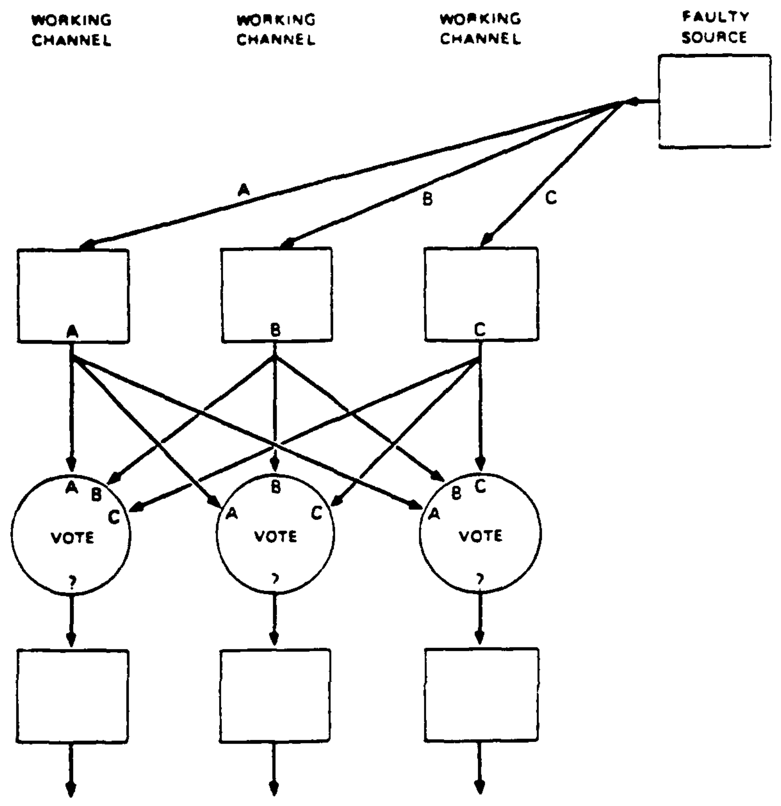


Figure 5.2: Distribution of Information from a Single Faulty Source to a Three-Channel System

and the system fails.

Figure 3 shows a three channel system with two working and one faulty channels. Here information present in just one of the channels is to be distributed to all three channels and be used in a replicated calculation. The faulty source distributes different values to the two working channels, and compounds the problem by repeating the same erroneous values (suitably transformed if necessary) in the next, voted, stage of the system. Note that not only do the two working channels continue to receive inconsistent values, even after voting, but also each of the two working channels can be mislead into believing that it is the other working channel that is faulty.

The existence of this problem was discovered during the design of SIFT, a reliable aircraft control system, and is discussed in Pease et al., JACM April 1980, where it is shown that no solution is possible in a purely three channel system. An algorithm, called the interactive consistency algorithm, is given for a four channel system containing a single faulty channel, and extended to the masking of N faults in a 3N+1 channel system.

The basic interactive consistency algorithm is given in Figure 4. One of the four channels is the single point source of the information, and the three other channels are used to replicate that information. Once the information is replicated, any or all of the channels can vote the replicated information with confidence that all voters in working channels will produce the same majority value, or alternatively all working voters will find no majority and will return a default value. For this algorithm to be effective against all faults, the channel that is the source of the information must be distinct from the three channels that perform the replication.

Consider the possibility that the source channel is faulty. It may then distribute different values to the other channels. The three replicating channels must all be working, and thus every working voter must get the same set of inputs. If at least two of the replicating channels have the same value, every working voter will find that value as its majority, while if all three replicating channels have different values, every working voter will return the default value. (If the source is faulty, the interactive consistency algorithm cannot of course guarantee a correct value from that source, but only a value that is consistent across all working channels.)

Consider the possibility that one of the three replicating channels is

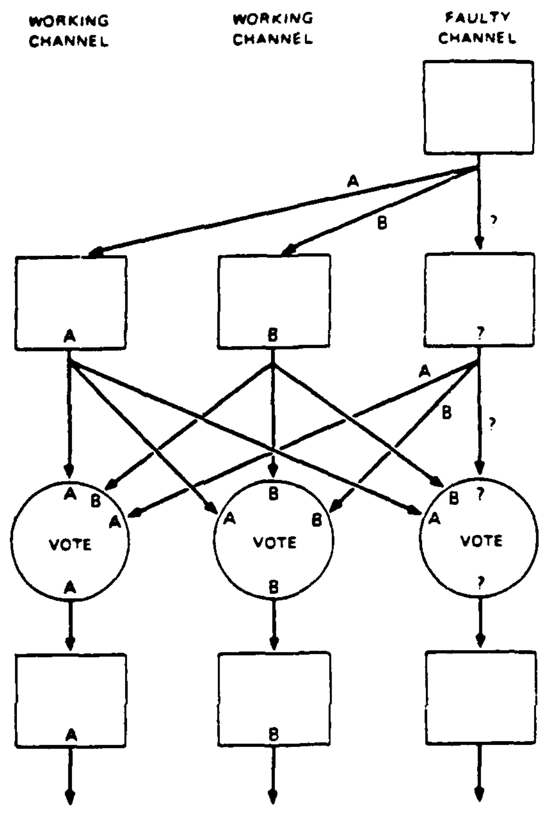


Figure 5.3: Distribution of information from a Single Channel to Three Channels

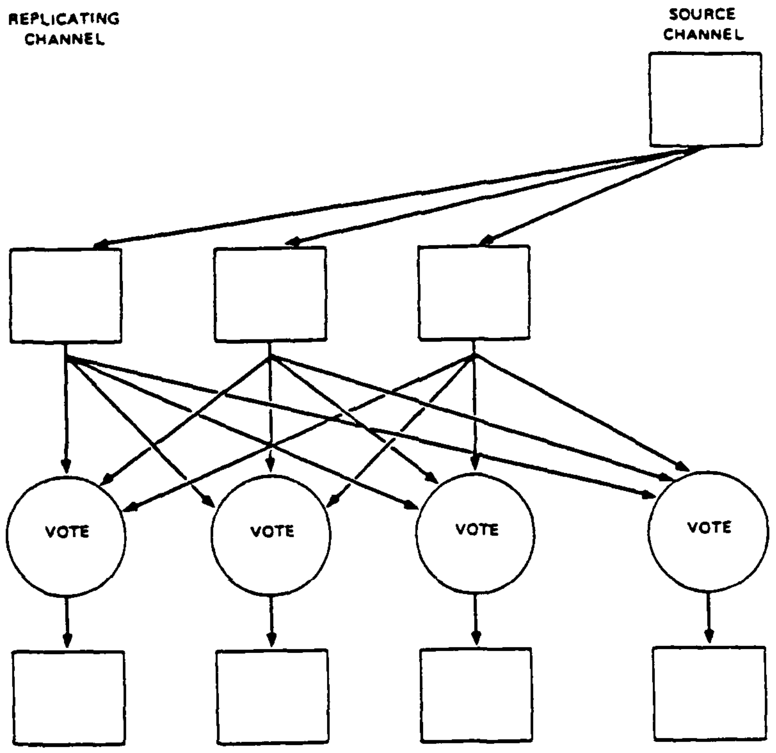


Figure 5.4: The Interactive Consistency Algorithm

faulty. Now the source is necessarily working and will distribute the same correct value to each of the two working replicators, which will replicate it. Thus each working voter obtains at least two correct inputs and is able to produce the correct value as its result.

In SIFT, four circumstances were found in which a value from a single source had to be distributed to three replicated channels, namely:

- input from a sensor
- error reports from a voter
- interfaces between unreplicated and replicated tasks
- synchronization of processor clocks.

The first three of these require the use of the interactive consistency algorithm to protect the system against malicious faults. The fourth is of special interest in that exact agreement is not necessary for clock synchronization, and thus slightly simpler algorithms guaranteeing approximate agreement suffice.

5.3 Maintenance of Approximate Consistency

In SIFT, as in many other fault tolerant systems, each processor has its own clock and operation of the system depends on these clocks remaining synchronized (to within 50ms in SIFT). Many prior systems used three channels, three clocks, and a clock synchronization algorithm based on each clock synchronizing itself periodically to the median clock of the three. It is instructive to consider why this "obviously sound" approach is invalid.

Figure 5 shows a system with two working clocks (A and B) and a faulty clock (C). We may assume that clock A runs slightly faster than clock B. Clock C presents to clock A an erroneous clock value indicating that clock C is running faster even than clock A, causing clock A to assume that it is the median clock. Thus clock A makes no correction to its value. Similarly, clock C presents to clock B a value indicating that it is behind even clock B, causing clock B to assume that it is the median clock and make no correction to its clock value. By this strategy, the faulty clock C

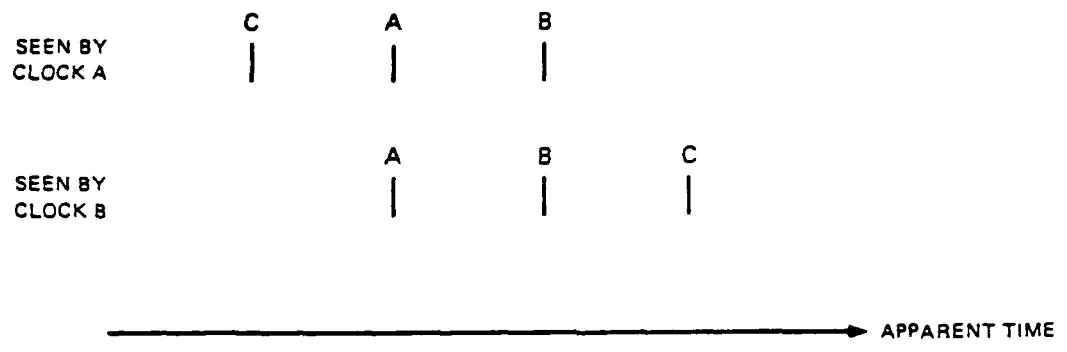


Figure 5.5: A Failure Mode of the Median Clock Synchronization Algorithm

can induce clocks A and B to operate without correcting their clock values as they gradually drift apart until the system fails. Single point component faults that could cause this "malicious" behavior have been found even in purely analog clock systems.

It is tempting to attempt minor corrections to the three channel clock synchronization algorithms, aimed at preventing this behavior. As yet we have no rigorous mathematical proof that no three channel algorithm can exist, but we believe that the approximate agreement needed for clock synchronization requires the same number of channels as the exact agreement discussed above.

In SIFT, a four channel clock synchronization algorithm is used in which each clock is periodically resynchronized to the mean of the four clocks. To protect against wildly erroneous clock values, the algorithm imposes a bound within which a clock value must lie to be included in the averaging calculation. For n processors of which at most m are faulty, with R as the resynchronization interval and S as the time taken for resynchronization, and if ϵ is the maximum clock reading error and ρ the maximum rate of clock drift, it can be shown that the maximum skew between working clocks will not exceed

$$\frac{n}{(n-3m)}(2\epsilon + \rho(R + \frac{2(n-m)S}{n})).$$

A similar problem has been examined by L. Webster in closed loop control systems. He found that use of a median voting algorithm in a three channel system favors the median channel, effectively disconnecting the two other channels from the closed loop. Without cross coupling between the integrators of the three channels, this results in uncontrolled accumulation of error terms in the integrators of two of the channels, rendering them useless for error masking. With cross coupling, the integrators are vulnerable to precisely the same problem as the clocks above.

The possibility of failure to maintain approximate consistency appears to exist in any three channel system containing embedded integrators.

5.4 Asynchronous Multichannel Systems

Existing fault tolerant multichannel systems using forward error correction, whether majority voted or dual-dual, depend on an exact equality between the result values of the various channels. To ensure this exact equality of their outputs, the various channels must all perform exactly the same calculation on exactly the same input values at approximately the same time. This exposes such systems to an unquantifiable risk of correlated faults generating errors simultaneously in several channels. Such correlated faults might result from some external influence, such as lightning or cosmic rays, or from accumulation of latent faults not within the coverage of the diagnostics, or from design faults in the hardware logic or the software.

A much higher degree of confidence in the resilience of the system to correlated faults would result from a system design in which each channel performs its calculation at different times, on different input values, and obtains different outputs. It is even possible to consider the use of different algorithms in each of the channels. Unfortunately, as exhibited above, without an exact match between channels, standard voting techniques are vulnerable to faults that cause loss of consistency between channels and

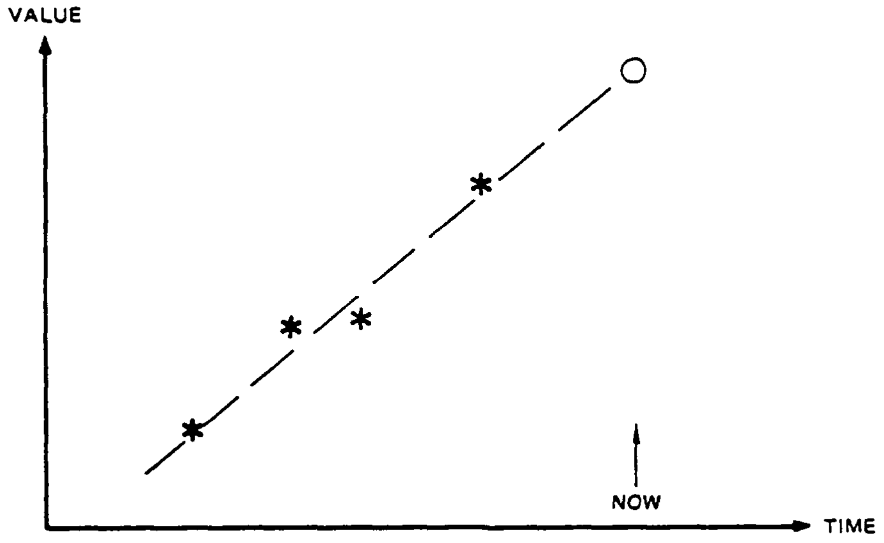


Figure 5.6: Extrapolation from Past Values to a Most Probable Current Value

thus system failure. We seek here to provide alternative algorithms that permit differences between channels without risk of loss of consistency.

The first thoughts on an approach to such asynchronous error masking envisage a system of four channels. Each channel operates at the required iteration rate but completely unsyschronized with the other channels, thus minimizing interaction between channels. Each result produced would carry a timestamp. A processor, when voting such a result, would have access to the four most recent values, one from each channel, together with their timestamps. From these it would be possible to extrapolate to a most probable current value, as shown in Figure 6.

More formally, if $R_{i,p}$ is the i'th broadcast result from processor p, containing a value $v_{i,p}$ and a timestamp $t_{i,p}$, and if the most recent result so far received from processor p is n_p, the algorithm can be expressed as:

$$\text{consensus value} = F(v_{n_a,a}, t_{n_a,a}, v_{n_b,b}, t_{n_b,b}, v_{n_c,c}, t_{n_c,c}, v_{n_d,d}, t_{n_d,d})$$

where F is some function to be determined, and a, b, c, d are the four processors.

Unfortunately, it is easy to show that the timestamps do not assist in the maintenance of consistency in the absence of any constraints on the times at which results are calculated. If greater weight is given to more recent values, those values may be erroneous values increasing the vulnerability of the system. In particular, consider the case in which three good values are reported approximately simultaneously and subsequently an erroneous value is reported. Any preference given to recent values can only render the consensus less reliable than that obtained by ignoring the timestamps.

Consideration can also be given to the clock synchronization algorithm described above. Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

```
**For** $i$ **in** $b, c, d$ : $v'_i$ =**if** $v_i > v_a + \delta$ **or** $v_i < v_a - \delta$
                    **then** $v_a$
                    **else** $v_i$

**and then**: consistent result = $\frac{v_a + v'_b + v'_c + v'_d}{4}$
```

That algorithm does indeed maintain consistency between channels, but the rate of convergence is very weak and the drift and error signals that can be introduced by undetected faulty clocks are much larger than the permitted drift and jitter of working clocks. In the clock synchronization application this is not critical for the individual clocks have performance characteristics much better than those required for typical system applications. For a control system application however, the errors introduced by a faulty channel can easily overwhelm the control action of the system, and thus such an algorithm is clearly unacceptable.

A possible alternative approach requires that the four channels compute their results at uniform phases within the iteration interval, one channel generating a value at the start of the interval, a second channel generating its result a quarter of the interval later, etc., as shown in Figure 7. This additional information allows the algorithm an improved ability to compute a most probable current value and to reject erroneous values. The uniform spacing at which results are generated through the interval greatly simplifies calculations compared with a system in which such spacings are arbitrary, and thus assists in reducing the voting calculation overhead.

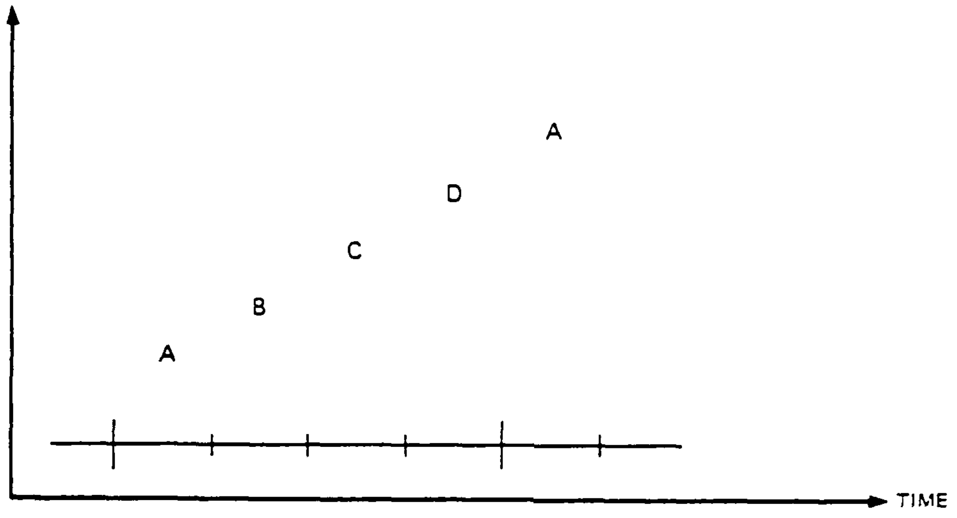


Figure 5.7: Calculation of Results at Uniform Phases within an Interval

An initial evaluation of such a system, using the arithmetic mean of the four values for the most probable current value, as in the clock synchronization algorithm. Each channel uses fixed limits for the acceptable deviation of the values computed by other channels from its own most recent value, but those limits can differ for each of the other channels. Thus if δ is an appropriate acceptable deviation for the channel whose result was computed one quarter of an iteration later, then 1.3δ is an appropriate limit for the channel computing half an iteration later and 1.2δ for the channel computing three quarters of an iteration later.

Here, if processor a is considering the values generated by processors b, c, d, with current values v_a, v_b, v_c and v_d,

```
$v'_b =$ if $v_b > v_a + \delta$ or $v_b < v_a - \delta$
    then $v_a$
    else $v_b$
$v'_c =$ if $v_c > v_a + 1.3\delta$ or $v_c < v_a - 1.3\delta$
    then $v_a$
    else $v_c$
```

```
$v'_d =$ **if** $v_d > v_a + 1.2\delta$ **or** $v_d < v_a - 1.2\delta$
    **then** $v_a$
    **else** $v_d$
```

and then: consistent result $= \frac{v_a + v'_b + v'_c + v'_d}{4}$

Unfortunately, while this algorithm appears to be better than the basic clock synchronization algorithm, it is only slightly so and the drift and error signals introduceable by a fault are still at least comparable to the maximum permissible control action of the system. Thus the algorithm is still unacceptable.

We can refine the algorithm by giving different weights to each of the values, for instance:

$$\text{consistent result} = \frac{v_a + 2v'_b + 3v'_c + 4v'_d}{10}$$

but the effect is marginal and still far from providing acceptable margins for control purposes.

Error masking algorithms such as these act as filters and, like all filters, necessarily introduce delay into the control loop. The algorithms above introduce a delay of about 2/3 of an iteration. To maintain the same margins of loop stability, the introduction of such a delay would require an increase in the iteration rate of about 33%.

A number of possible improvements to the algorithm are under consideration. We are currently working on algorithms that make better use of the relative timing of results, both by giving greater weight to more recent results in estimating the most probable current value, and also by considering the values generated by other channels when determining the acceptability of a result. A further possibility is the use of a five channel system fully capable of rejecting the most malicious faults which degrades on the first reconfiguration to a four channel system capable of rejecting all faults except those malicious faults in which different information is delivered to different destinations by the broadcast mechanisms. Since the probability of a second fault during a mission is low, and the probability of a malicious fault is also low, such a system might be judged to be adequately reliable.

References

[1] A. Mazurkiewicz. *Semantics of Concurrent Systems: A Modular Fixed Point Trace Approach.* Technical Report 84-19, Institute of Applied Mathematics and Computer Science, University of Leiden, 1984.

[2] W. Brauer, editor. *Net Theory and Applications.* Springer-Verlag, Berlin, 1980.

[3] P. J. Courtois, F. Heymans, and David L. Parnas. Concurrent control with "readers" and "writers". *Communications of the ACM,* 14(10):190-199, October 1971.

[4] Leslie Lamport. Concurrent reading and writing. *Communications of the ACM,* 20(11):806-811, November 1977.

[5] Leslie Lamport. The mutual exclusion problem. To appear in *JACM.*

[6] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems,* 1(1):84-97, July 1979.

[7] Leslie Lamport. Time. clocks and the ordering of events in a distributed system. *Communications of the ACM,* 21(7):558-565, July 1978.

[8] Leslie Lamport. What it means for a concurrent program to satisfy a specification: why no one has specified priority. In *Proceedings of the Twelfth ACM Symposium on Principles of Programming Languages,* ACM SIGACT-SIGPLAN, New Orleans, January 1985.

[9] Peter E. Lauer, Michael W. Shields, and Eike Best. *Formal Theory of the Basic COSY Notation.* Technical Report TR143, Computing Laboratory, University of Newcastle upon Tyne, 1979.

[10] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, Berlin, 1980.

[11] Gary L. Peterson. Concurrent reading while writing. *ACM Transactions on Programming Languages and Systems*, 5(1):46–55, January 1983.

[12] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th Symposium on the Foundations of Computer Science*, ACM, November 1977.

[13] Glynn Winskel. *Events in Computation.* PhD thesis, Edinburgh University, 1980.

Appendix

Proof of Proposition 1

It follows from (1) that, for any operation execution A in S, the relations $\longrightarrow$ and $\dashrightarrow$ are not changed by either of the following two changes to the global-time model, where $\delta > 0$:

1. Changing s_A to $s_A - \delta$ if, for all $B \in S$: $f_B < s_A$ implies $f_B < s_A - \delta$.
2. Changing f_A to $f_A + \delta$ if, for all $B \in S$: $f_A < s_B$ implies $f_A + \delta < s_A$.

Let T denote the set of numbers s_A and f_A for all A in S, and for any real t, let $S(t) = \{r \in T : r < t\}$ and $F(t) = \{r \in T : r > t\}$. M2 implies that for any t, $\max S(t) < t$ and $t < \min F(t)$.

For any A, if s_A equals s_B or f_B for some $B \neq A$, I can change s_A to $s_A - \delta$, where $0 < \delta < \epsilon$ is chosen so that $s_A - \delta > \max S(s_A)$. Similarly, if f_A equals s_B or f_B for some $B \neq A$, I can change f_A to $f_A + \delta$, where $0 < \delta < \epsilon$ and $f_A + \delta < \min F(s_A)$.

The details of the formal proof, which involves an inductive definition of s' and f' based upon the countability of S, is left to the reader.

Proof of Propositions 2 and 3

The "only if" part of Proposition 2 follows immediately from (1). To prove Proposition 3 and the "if" part of Proposition 2, I prove that for every system execution $S, \longrightarrow, \dashrightarrow$ there exists a global-time model s, f such that for every $A, B \in S$:

- $A \longrightarrow B$ implies $f_A < s_B$
- $A \dashrightarrow B$ implies $s_A < f_B$

The relations $\overset{\prime}{\longrightarrow}$ and $\overset{\prime}{\dashrightarrow}$ defined by this global-time model satisfy the requirements of Proposition 3. Moreover, if $S, \longrightarrow, \dashrightarrow$ satisfies A#, then $\overset{\prime}{\dashrightarrow}$ must equal $\dashrightarrow$, since if A# holds then $A \not\dashrightarrow B$ implies $B \longrightarrow A$, which implies $B \overset{\prime}{\longrightarrow} A$, so $A \overset{\prime}{\not\dashrightarrow} B$, and $A \not\longrightarrow B$ implies $B \dashrightarrow A$, which implies $B \overset{\prime}{\dashrightarrow} A$, so $A \overset{\prime}{\not\longrightarrow} B$.

The following proposition is used in this proof and in a later one.

Proposition 10 *Let T be the set consisting of all elements of the form s_A and f_A for $A \in S$ (the elements of T are uninterpreted symbols, not necessarily real numbers), and let $\prec$ be the smallest transitively closed relation such that*

- *If $A \longrightarrow B$ then $f_A \prec s_B$.*
- *If $A \dashrightarrow B$ or $A = B$ then $s_A \prec f_B$.*

Then $\prec$ is an irreflexive partial ordering.

Proof: Define the relations $\xrightarrow{o}$, $\xrightarrow{s}$, and $\xrightarrow{d}$ on T as follows:

- For all A: $s_A \xrightarrow{o} f_A$.
- $f_A \xrightarrow{s} s_B$ if and only if $A \longrightarrow B$.
- $s_A \xrightarrow{d} f_B$ if and only if $A \dashrightarrow B$.

Let $\longrightarrow$ be the union of the three relations $\xrightarrow{o}$, $\xrightarrow{s}$, and $\xrightarrow{d}$, so $\prec$ is the transitive closure of $\longrightarrow$. It suffices to prove that $\longrightarrow$ is an acyclic relation.

The proof is by contradiction. Choose a shortest cycle formed by the $\longrightarrow$ relation. A cycle composed entirely of $\xrightarrow{o}$ and $\xrightarrow{s}$ relations would violate A1, so the cycle must contain a portion of the form:

$$f_A \xrightarrow{s} s_B \xrightarrow{d} f_C \xrightarrow{s} s_D$$

since $\xrightarrow{s}$ is the only relation from an f to an s and there are no s to s or f to f relations. I can apply A4 to deduce that $f_A \xrightarrow{s} s_D$, which contradicts our assumption that the cycle had minimai length, proving Proposition 10. ∎

Returning to the proof of Propositions 2 and 3, we see that $\prec$ is an irreflexive acyclic relation. Moreover, A5 implies that for any $t \in T$, $t \prec s$ for all but a finite number of elements s. This, together with the countability of T, implies that $\prec$ can be completed to a total ordering $<$ such that there is an order-preserving isomorphism of T with a subset of the natural numbers. Identifying the elements of T with the corresponding natural numbers provides the desired global-time model.

Proof of Proposition 4

Let T be the set of all numbers s_A and f_A for $A \in S$, and let $\prec$ be the partial ordering on T defined as in Proposition 10 for the precedence relations $\xrightarrow{l}$ and $\dashrightarrow^{l}$, namely, the smallest partial order such that $A \xrightarrow{l} B$ implies $f_A \prec s_B$, and $A \overset{l}{\dashrightarrow} B$ or $A = B$ implies $s_A \prec f_B$. Observe that the following hold for all A and B in S:

(a) Either $s_A \prec f_B$ or $f_B \prec s_A$ (by A#).

(b) $f_A < s_B$ implies $f_A \prec s_B$ (by H3).

To prove the proposition, it suffices to construct s', f' such that[5] $s \le s' \le f' \le f$ and for all A and B: $f_A \prec s_B$ implies $f'_A < s'_B$ and $s_A \prec f_B$ implies $s'_A < f'_B$.

Let s', f' be any global model satisfying

$$f'_A < s'_B \text{ implies } f_A \prec s_B \tag{5}$$

The pair of operation executions A, B is said to be *out of order for* s', f' if $f_A \prec s_B$ and $s'_B < f'_A$. It follows from (a) and (b) that if there are no out-of-order pairs, then s', f' satisfies the conditions of the proposition.

I will construct s', f' inductively by constructing a sequence of nondegenerate models s^i, f^i with $s^i \le s^{i+1} \le f^{i+1} \le f^i$ having s^0, f^0 equal to s, f and s', f' equal to their limit. This is done by first choosing the enumeration of all out-of-order pairs of s, f such that, for any subset of them, the minimal element is the one A, B having the smallest value of f_A and, among all such pairs A, B', the one having the largest value of s_B. It follows from M2 that such a minimal element exists for any nonempty set, so this defines an enumeration of the out-of-order pairs of s, f.

If A, B is the ith out-of-order pair, then s^i, f^i will be defined to be the same as s^{i-1}, f^{i-1} except that $s_B^{i-1} < f_A^i < s_B^i < f_A^{i-1}$. This implies that the set of out-of-order pairs for s^i, f^i equals the set of out-of-order pair for s^{i-1}, f^{i-1} minus the pair A, B. Moreover, it follows from A5 and (b) that any operation execution belongs to only a finite number of out-of-order

[5] I employ the usual notation that for functions f and g with the same domain, $f \le g$ if and only if $f(x) \le g(x)$ for all x in their domain.

pairs of s, f, so the limit s', f' of the models s^i, f^i exists, satisfies (5), and has no out-of-order pairs, proving the proposition.

For notational convenience, the construction of s^i, f^i from s^{i-1}, f^{i-1} is given for the case $i = 0$. So, I assume that s, f satisfies (b), which is the same as (5), and has a minimal out-of-order pair A, B. I construct s^1, f^1 by decreasing f_A and increasing s_B to get $f_A^1 < s_B^1$, without creating any new out-of-order pairs. (The construction for any i is the same except with more superscripts.)

Let X be the operation execution with the largest value of s_X such that $s_X \prec f_A$; if there is no such X, let $s_X = -\infty$. It follows from (b) and the nondegeneracy of s, f that $s_X < f_A$. Observe that there is no C with s_C in the interval $(\max(s_X, s_B), f_A]$, since, by choice of s_X, this would imply $f_A \prec s_X$, which would contradict the maximality of s_B. Therefore, if I define f_A^1 to be $\max(s_X, s_B)^+$, then s, f^1 satisfies (5) and has the same set of out-of-order pairs as s, f, where t^+ denotes a value larger than t such that there is no value s_C or f_C in the interval $(t, t^+]$.

If $s_B > s_X$, so $f_A^1 = s_B^+$, then I can define s_B^1 to be $(f_A^1)^+$ and it is clear that s^1, f^1 also satisfies (5) and has the same set of out-of-order points as s, f^1 except that A, B is not out of order for s^1, f^1, so we are done.

Therefore, I need only consider the case $s_B < s_X$. (Since $s_X \prec f_A$, we must have $s_B \neq s_X$.) I claim that there is no f_C in the interval $[s_B, s_X]$. If there were, then (a) and (b) imply that $f_C \prec s_X$ and $s_B \prec f_C$, which, since $s_X \prec f_A$, would imply $s_B \prec f_A$, contrary to the assumption that A, B is out of order for s, f. Therefore, defining $s^{.5}$ to be the same as s except with $s_B^{.5} = s_X^+$, we see that $s^{.5}, f^1$ satisfies (5) and has the same set of out-of-order pairs as s, f^1. Replacing s by $s^{.5}$ and starting our argument again, we are in the case $s_X^{.5} < s_B^{.5}$ that was considered above. This completes the proof.

Proof of Proposition 5

If $\longrightarrow$ and $\dashrightarrow$ are any relations on a set S, let the *completion* of $\longrightarrow$ and $\dashrightarrow$ be the relations $\xrightarrow{'}$ and $\overset{'}{\dashrightarrow}$, where $\xrightarrow{'}$ is the smallest transitively closed extension of $\longrightarrow$ such that $A \xrightarrow{'} B \dashrightarrow C \xrightarrow{'} D$ implies $A \xrightarrow{'} D$, and $\overset{'}{\dashrightarrow}$ is the union of $\dashrightarrow$ and $\xrightarrow{'}$. Thus, $A \xrightarrow{'} B$ if and only if there exists a chain

$$A = A_1 \Longrightarrow \cdots \Longrightarrow A_n = B$$

where $\Longrightarrow$ denotes either $\longrightarrow$ or $\longrightarrow C \dashrightarrow D \longrightarrow$ for some C and D.

Proposition 11 *If* $\longrightarrow$ *satisfies A5;* $\xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ *is the completion of* $\longrightarrow, \dashrightarrow$*; and* $\xrightarrow{\prime}$ *is acyclic; then* $\mathcal{S}, \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ *is a system execution.*

Proof: I must show that $\mathcal{S}, \xrightarrow{\prime}, \overset{\prime}{\dashrightarrow}$ satisfies A1–A5. The only nonobvious part is, in the proof of A2, showing that if $A \xrightarrow{\prime} B$ then $B \not\dashrightarrow A$. However, as observed above, this follows from A1 and A4. ∎

To prove Proposition 5, let $\xrightarrow{o}$ be the union of the relations $\xrightarrow{\bullet}$ and $\xrightarrow{\mathcal{H}}$, and let $\overset{o}{\dashrightarrow}$ be the union of $\overset{\mathcal{H}}{\dashrightarrow}$ and the restriction of $\xrightarrow{\bullet}$ to $\mathcal{T}$. Note that the restriction of $\xrightarrow{o}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$ (by H3). I define $\xrightarrow{\mathcal{HT}}, \overset{\mathcal{HT}}{\dashrightarrow}$ to be the completion of $\xrightarrow{o}, \overset{o}{\dashrightarrow}$.

I claim that to prove Proposition 5, it suffices to show that $\xrightarrow{\mathcal{HT}}$ is acyclic and the restrictions of $\xrightarrow{\mathcal{HT}}$ and $\overset{\mathcal{HT}}{\dashrightarrow}$ to $\mathcal{H}$ equal $\xrightarrow{\mathcal{H}}$ and $\overset{\mathcal{H}}{\dashrightarrow}$. Proposition 11 then implies that $\mathcal{H} \cup \mathcal{T}, \xrightarrow{\mathcal{HT}}, \overset{\mathcal{HT}}{\dashrightarrow}$ is a system execution, which is easily seen to be implemented by $\mathcal{S} \cup \mathcal{T}, \longrightarrow, \dashrightarrow$. (The definition of $\xrightarrow{\mathcal{HT}}$ and $\overset{\mathcal{HT}}{\dashrightarrow}$ implies that their restrictions to $\mathcal{T}$ are extensions of $\xrightarrow{\bullet}$ and $\overset{\bullet}{\dashrightarrow}$.)

Moreover, I claim that it suffices to prove that the restriction of $\xrightarrow{\mathcal{HT}}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$. It follows immediately from the definition of $\overset{\mathcal{HT}}{\dashrightarrow}$ and A2 that if the restriction of $\xrightarrow{\mathcal{HT}}$ equals $\xrightarrow{\mathcal{H}}$, then the restriction of $\overset{\mathcal{HT}}{\dashrightarrow}$ to $\mathcal{H}$ must equal $\overset{\mathcal{H}}{\dashrightarrow}$. Furthermore, the definition of the completion and the acyclicity of $\xrightarrow{\bullet}$ imply that any cycle of $\xrightarrow{\mathcal{HT}}$ relations must include an element of $\mathcal{H}$, so $A \xrightarrow{\mathcal{HT}} A$ must hold for some $A \in \mathcal{H}$. If the restriction of $\xrightarrow{\mathcal{HT}}$ to $\mathcal{H}$ equals $\xrightarrow{\mathcal{H}}$, then the acyclicity of $\xrightarrow{\mathcal{HT}}$ follows from the acyclicity of $\xrightarrow{\mathcal{H}}$. Thus, it suffices to prove that if $A \xrightarrow{\mathcal{HT}} B$ then $A \xrightarrow{\mathcal{H}} B$.

By definition of $\xrightarrow{\mathcal{HT}}$, if $A \xrightarrow{\mathcal{HT}} B$ then there exists a chain $A = A_1 \Longrightarrow \cdots \Longrightarrow A_n = B$, where $\Longrightarrow$ denotes either $\xrightarrow{o}$ or $\xrightarrow{o} C \overset{o}{\dashrightarrow} D \xrightarrow{o}$. Note that if A_i and A_{i+1} are both in $\mathcal{H}$, then $A_i \Longrightarrow A_{i+1}$ implies that $A_i \xrightarrow{\mathcal{H}} A_{i+1}$, and if they are both in $\mathcal{T}$ then $A_i \Longrightarrow A_{i+1}$ implies that $A_i \xrightarrow{\bullet} A_{i+1}$. Therefore, it suffices to show that any such chain that is of minimal length has length one.

If three consecutive elements A_i, A_{i+1}, and A_{i+2} in this chain are either all in $\mathcal{H}$ or all in $\mathcal{T}$, by the transitivity of $\xrightarrow{\mathcal{H}}$ and $\xrightarrow{\bullet}$ it follows that $A_i \Longrightarrow A_{i+2}$. Therefore, in a minimal-length chain, A_i must be in $\mathcal{H}$ if i is odd and in $\mathcal{T}$ if i is even. If $n > 0$, then we have $A_1 \Longrightarrow A_2 \Longrightarrow A_3$, with A_1 and A_3 in $\mathcal{H}$ and A_2 in $\mathcal{T}$. A $\xrightarrow{o}$ relation between an element of $\mathcal{H}$ and an element of $\mathcal{T}$ must be a $\xrightarrow{\bullet}$ relation. Considering the two possible cases for each $\Longrightarrow$ relation, using A1 and A4 for the relations $\xrightarrow{\bullet}$ and $\overset{\bullet}{\dashrightarrow}$, it follows from $A_1 \Longrightarrow A_2 \Longrightarrow A_3$ that $A_1 \xrightarrow{\bullet} A_2 \xrightarrow{\bullet} A_3$, so $A_1 \Longrightarrow A_3$. This contradicts the assumption of the minimality of n, proving that $n = 1$ and $A \xrightarrow{\mathcal{H}} B$, which completes the proof of the proposition.

Proof of Propositions 6 and 7

Parts (a) and (b) of Proposition 6 are immediate consequence of Definition 4. To prove part (c), observe that this definition implies $V^{[j]} \dashrightarrow v^{[i,j]}$. The result is immediate if $j = 0$. If $j > 0$, then $V^{[j-1]} \longrightarrow V^{[j]}$. Combining these two relations with the hypothesis, we have

$$V^{[j-1]} \longrightarrow V^{[j]} \dashrightarrow v^{[i,j]} \longrightarrow v^{[i',j']}$$

Axiom A4 implies that $V^{[j-1]} \longrightarrow v^{[i',j']}$, which, by A2, implies $v^{[i',j']} \not\dashrightarrow V^{[j-1]}$. This finishes the proof of Proposition 6.

To prove part (a) of Proposition 7, observe that it follows immediately from Definition 4 that $V^{[k]} \dashrightarrow R$ implies $k \leq j$. Conversely, I assume $k \leq j$ and show this implies $V^{[k]} \dashrightarrow R$. Since $V^{[j]} \dashrightarrow R$, the desired conculsion is immediate if $k = j$. If $k < j$, then $V^{[k]} \longrightarrow V^{[j]}$, and it follows from A3.

For part (b), Definition 7 implies that if $i < k'$ then $R \dashrightarrow V^{[k']}$. Letting $k' = k + 1$, this shows that if $i \leq k$ then $R \dashrightarrow V^{[k+1]}$. Conversely, suppose $R \dashrightarrow V^{[k+1]}$. Then $k+1 \neq i$. If $k+1 < i$, then $V^{[k+1]} \longrightarrow V^{[i]}$, so A3 would imply $R \dashrightarrow V^{[i]}$ contrary to Definition 4. Hence, we must have $i < k + 1$ so $i \leq k$, completing the proof of Proposition 7.

Proof of Propositions 8 and 9

Apply Proposition 3 to extend the given $\longrightarrow$ and $\dashrightarrow$ relations so they satisfy A#. It follows from B1 that this extension does not add any new precedence relations between reads and writes. A read sees $v^{[i,j]}$, as defined by these new relations, if and only if it sees $v^{[i,j]}$ in the original system execution. Hence, the new system execution, which satisfies A#, satisfies the hypotheses of the appropriate proposition. Applying Proposition 2, I can therefore assume a nondegenerate global-time model for the system execution.

For the proof of Proposition 9, let ϕ be the assumed function. For the proof of Proposition 8, ϕ is defined as follows. If R is a read that sees $v^{[i,j]}$, for a safe register define $\phi(R)$ to equal j, and for a regular register define it to be a value satisfying conditions 1 and 2 in the hypothesis of Proposition 9. (B4 implies that such a definition is possible.)

I first show that $S, \longrightarrow, \dashrightarrow$ (which I am assuming to have a nondegenerate global-time model) trivially implements a system execution in which reads are instantaneous, which is all that is required to prove Proposition 8. Given the nondegenerate global-time model s, f for $S, \longrightarrow, \dashrightarrow$, it suffices to find a global-time model s', f' with $s \leq s' \leq f' \leq f$ in which all reads are instantaneous, such that B1–B4 hold for the system execution defined by s', f'.

For notational convenience, let s_i and f_i denote $s_{V^{[i]}}$ and $f_{V^{[i]}}$, respectively. Let s', f' be the same as s, f except that, for a read R, define s'_R to equal the maximum of the following three quantities:

- s_R
- $(s_{\phi(R)})^+$
- $\max\{s_{R'} : \phi(R') < \phi(R) \text{ and } s_{R'} < f_R\}^+$

and define f'_R to equal $(s'_R)^+$. When the appropriate careful definition of t^+ is given, this results in a nondegenerate global-time model in which every read is instantaneous. I must check that, for any read R: $s_R \leq s'_R \leq f'_R \leq f_R$, B1–B3 remain satisfied, and B4 remains satisfies when v is regular.

It is immediate by the definition of s'_R that $s_R \leq s'_R$. Since $f'_R = (s'_R)^+$, to establish the remaining inequalities, I need to show that $f'_R < f_R$. If R sees $v^{[i,j]}$, then, by Definition 4, $s_j < f_R$ (the strict inequality comes from nondegeneracy), and, since $\phi(R) \leq j$, $s_{\phi(R)} < f_R$. The required inequality now follows easily from the definition of s'_R.

I must now show that B1–B3 and, if v is regular, B4 hold for the new precedence relations. B1 and B2 are trivial. For B3 and B4, consider what a read sees in the new system execution if it sees $v^{[i,j]}$ in the original one. There are three cases:

1. If $f_{\phi(R)} < s_R$ then
   (a) if $s_R < s_{\phi(R)+1}$ then R sees $v^{[\phi(R),\phi(R)]}$
   (b) if $s_{\phi(R)+1} < s_R$ then R sees $v^{[\phi(R),\phi(R)+1]}$
2. If $s_R < f_{\phi(R)}$ then R sees $v^{[\phi(R)-1,\phi(R)]}$.

Moreover, it is immediate from Definition 4 that case 1(b) is impossible if $\phi(R) = j$, which is the case when v is assumed to be only safe. This definition also implies that $f_j < s_R$ if and only if $i = j$. Thus, when v is only safe, R sees $v^{[i,i]}$ in the new system execution if and only if it does in the old, proving B3. For the case when v is regular, B3 and B4 follow immediately from the fact that R returns the value $v^{[\phi(R)]}$. This finishes the proof of Proposition 8.

To complete the proof of Proposition 9, I first show that if $\phi(R) < \phi(S)$ for reads R and S, then $f'_R < s'_S$. The third hypothesis about ϕ implies that if $\phi(R) < \phi(S)$, then $s_R < f_S$. By the definition of s'_S, this implies that s'_S is greater than each of the three quantities of which s'_R is the maximum, so $s'_R < s'_S$. Since reads are instantaneous with respect to s', f', this implies $f'_R < s'_R$.

I must construct a new global-time model s'', f'', in which writes are also instantaneous and B1–B3 are still satisfied, so that s'', f'' is the same as s', f' except for writes, and for any write $V^{[k]}$: $s'_k \leq s''_k \leq f''_k \leq f'_k$. (Note that B5 follows from the fact that reads and writes are instantaneous, and B4 follows from B3 and B5.)

Let s''_k be the maximum of the two quantities s'_k and $\max\{f'_R : \phi(R) = k-1\}^+$, and let f''_k be $(s''_k)^+$. Since $v^{[\phi(R)]}$ is one of the values "seen" by R in the system execution defined by s', f', if $\phi(R) = k-1$ then $s'_R < f'_k$, which implies that $s''_k < f'_k$. We therefore have $s' \leq s'' \leq f'' \leq f'$, and reads and writes are both instantaneous in s'', f''. Again, B1 and B2 are trivial, so I need only prove B3.

Since reads and writes are instantaneous, B5 holds—a read R sees $v^{[i,i]}$; I must show that $i = \phi(R)$. The definition of s'' implies that $f''_R = f'_R < s''_{\phi(R)+1}$. I must therefore show that $s''_{\phi(R)} < s'_R$. In the global-time model s', f', the read R "sees the value" $v^{[\phi(R)]}$, so $s'_{\phi(R)} < s'_R$. By definition of s'', we can have $s''_{\phi(R)} > s'_R$ only if there exists some R' with $\phi(R') < \phi(R)$ and $f_{R'} > s'_R$. However, I showed above that $R' < R$ implies $f'_{R'} < s'_R$, which completes the proof.

Part VI

Experimental Implementation and Evaluation of the Trans Broadcast Protocol

6.1 Introduction

An earlier section of this report (Part III) introduced a novel link-level protocol for broadcast environments. The protocol, known as TRANS, exploits the characteristics of broadcast communications media in order to achieve reliable communication with minimal overhead.

This section of the report describes a prototype implementation of TRANS, which was undertaken so that the design and performance of the protocol could be evaluated. A great deal was learned about the behavior of the protocol during this process, including subtle problems in its design. However, this experimental implementation was undertaken towards the very end of the project, when time and funds were almost exhausted, and we were therefore unable to completely resolve some difficulties in the design of TRANS, or to collect as much data on its performance as we would have wished.

We believe that the subtle problems and difficulties encountered in the implementation of TRANS vindicate the decision to undertake that implementation. Protocols are notoriously difficult to get right, and claims based on only informal specifications and correctness arguments (as was the case with the previous description of TRANS) should be viewed with skepticism. The problems discovered in TRANS do not appear major and we believe they can be corrected. Unfortunately, there simply was not enough time to address them during this contract. The performance measurements that we were able to make are encouraging and suggest that broadcast protocols such as TRANS offer useful benefits in certain situations.

This experimental implementation and evaluation of TRANS has suggested several directions for future research. An increased understanding of the protocol has indicated several modifications that would lead to improved performance. It has become clear that the protocol should be specified and proved formally, and that implementation considerations must be addressed. Additional performance measurements are required to complete the evaluation of the protocol and comparisons should be made against alternative approaches. There are also several extensions to the protocol that can be examined. Finally, the protocol can be used as the basis for the design, implementation, and evaluation of a variety of distributed systems algorithms.

6.2 Specification of the TRANS Protocol

The starting point for our implementation of TRANS is the description of the protocol given in Part III of this report. However, rather than simply reproducing that description here, it will be useful to first provide some additional motivation and discussion.

The context in which TRANS is to operate assumes a communications medium using physical broadcast (such as Ethernet, or packet radio), and an applications environment that requires reliable broadcast communications. Conventional protocols that assume point to point communications could require a minimum of $2 \times (n-1)$ messages to transmit a message from one of n hosts to all the others (composed of $n-1$ individual transmissions from the sender to each recipient, and the same number of acknowledgments). A protocol that allows broadcast transmission but that requires individual acknowledgments could reduce this to n messages (1 transmission and $n-1$ acknowledgments).

If we are prepared to wait for acknowledgments until receiving hosts have messages of their own to transmit, then no additional messages may be required beyond the initial broadcast: receiving hosts simply save up acknowledgments and append them to their own messages. Assuming a community of n hosts all broadcasting at approximately the same rate, this could require each host to append an average of $n-1$ acknowledgments to each of its own messages.

The novel contribution of TRANS is that it attempts to reduce the number of acknowledgments that must be appended to each message by exploiting the broadcast character of the communications medium and the transitivity of acknowledgments.[1] If a host needing to acknowledge a message Y sees another message X carrying an acknowledgment for Y, then it need not acknowledge Y explicitly: its acknowledgment of X will *implicitly* acknowledge receipt of Y. Under favorable circumstances, this could result in each host having to explicitly acknowledge only 1 message in each of its own messages—the remaining $n-2$ being acknowledged implicitly. This can significantly reduce the bandwidth needed for a given degree of communication. In addition, it can significantly reduce the amount and frequency of communications required from individual hosts. This could

[1]The name of the protocol is derived from TRANSitivity.

be beneficial in packet radio situations, for example, where certain stations are attempting to operate under near radio-silence.

The naive protocol outlined above must obviously be modified to deal with the circumstance where a host fails to receive a message. Accordingly, *negative* acknowledgments are introduced so that hosts can indicate such failures. Henceforth, a (positive) acknowledgment will be referred to as an ack, while a negative acknowledgment is called a nack. (Machines will be referred to as *hosts*, although the earlier section on TRANS refers to them as nodes.) A host should append a nack to its next message if it receives a message in a corrupted state (but is able to recover the identity of the message), or learns—through the presence of acks on other messages—of the existence of a message that it has not received. Such nacks provoke the sender of the message concerned to retransmit it. A host that has a pending nack can discard it if it sees another message carrying a nack for the same message, since that prior nack will already be sufficient to provoke the retransmission that is desired.

Although the incorporation of nacks into the protocol may seem a small change, concerned solely with *liveness*, it turns out to greatly complicate the "reception analysis" component of the protocol.

This problem can be seen in the example shown in Figure 6.1. In this

```
          Z
   nack / \  ack
       /   \
      Y     X
   ack \   / ack
        \ /
         W
```

Figure 6.1: Difficulty Introduced by nacks

and subsequent similar figures, the named nodes (X, Y, Z etc.) represent messages, and the arcs between them represent (some of the) acks and nacks carried by the message at the *bottom* of the arc. The time dimension runs down the page, so the example in Figure 6.1 indicates that messages Y and X were sent sometime later than Z, and that Y nacked Z while

X acked it. Message W carried acks for both X and Y. The question is: can we deduce whether or not the host that sent W saw message Z? The answer is that it is very difficult to make such deductions in the presence of nacks. Suppose the sender of W did see Z, and that it then saw X. Since X carries an ack for Z, the sender of W will discard its own ack for Z and acknowledge it implicitly in its ack for X. On the other hand, if we assume that the sender of W did not see Z, then exactly the same argument applies *mutatis mutandis* with respect to Y and nacks. It might seem that this ambiguity could be resolved if the sender of W were not so hasty to discard its own pending ack for Z: then it could *explicitly* ack Z once it saw the nack carried by Y. A little thought will show that this stratagem cannot be relied upon. Consider the situation pictured in Figure 6.2. Here,

```
        Z
        |\  nack
        | \
    ack |  X
        | / nack
        |/
        Y
        |
    ack |
        |
        W
```

Figure 6.2: Further Difficulty Introduced by nacks

the host that sent W (which is assumed to have seen Z) might have been prepared to directly ack Z had it known of the ambiguity introduced by the nack carried by X, but it may not itself have seen X (the nack carried by Y will have caused it to discard its own nack for X) and may therefore be unaware of the nack which X carries for Z.

These examples show that a nack introduces uncertainty as to whether any messages further along an ack chain have been seen or not. Thus there is little point in retaining acks for messages that others have nacked—and so the TRANS protocol discards both pending nacks and acks whenever a nack is seen for the message concerned.

The previous discussion should have motivated the essential components of the TRANS protocol, whose description from Part III of this report is repeated below:

- Each message is broadcast with a header in which there is a message identifier containing the source of the message and a message sequence number. A version number is also included in the identifier to distinguish retransmissions. Sequence numbers can recycle over some suitably long interval. Each message also carries with it acknowledgments (positive and negative) to previous messages, and an error detecting code. Other fields in the header, such as a message destination list (for multicast), may be present but do not play any part in this protocol.

- Each node maintains a list of positive and negative acknowledgment message identifiers. Whenever it broadcasts a message, it appends this list of acknowledgments to the message, and then clears its list.

- When a node receives a message it has not previously received in an uncorrupted state, it adds the identifier as an acknowledgment to its list. If the message is uncorrupted, the identifier is added as a positive acknowledgment; if the message is corrupted, but with an uncorrupted header, the identifier is added as a negative acknowledgment.

- When a node sees a positive acknowledgment appended to a message that it receives, it deletes from its own list any positive acknowledgment for that message. When it sees a negative acknowledgment for a message, it deletes from its list *any* acknowledgment for that message, whether positive or negative.

- When a node sees a positive acknowledgment for a message that it has not received, it adds a negative acknowledgment to its list.

- If a node has no messages pending, it may be necessary to construct a null message to carry acknowledgment messages. The acceptable delay before transmitting a null message may differ for positive and negative acknowledgments.

- When a node receives a negative acknowledgment for one of its messages, or has received no positive acknowledgment within some time interval, it retransmits the message. The retransmission must be *identical* to the prior transmission, and thus must carry with it exactly the same acknowledgments, positive and negative, carried by the prior transmission of that message.

That part of the TRANS protocol described above is called *transmission control.* Transmission control is the set of rules used by a host to decide which acknowledgments are required and when it should reissue messages. One of the main functions of transmission control is to ensure *liveness*: a message must be retransmitted whenever there is doubt that it has been received by all hosts. The task of determining whether all hosts have definitely received a particular message (so that the sending host may take the irrevocable step of discarding the message) is the responsibility of a companion algorithm known as *reception analysis.* Although they appear to be separate, the transmission control and reception analysis algorithms must cooperate in order for the protocol to be implemented correctly and efficiently. Transmission control must cause all messages to be reliably delivered to all hosts. It must also provide enough information in the message traffic to permit reception analysis to be performed correctly. In particular, messages cannot be removed before they have been received everywhere and they should be removed as soon as they have been received everywhere. It would be advantageous if the information in the message traffic also allowed reception analysis to be performed very efficiently.

The reception analysis algorithm for TRANS is based on Theorem 2 of Part 5. The statement of that Theorem given in the earlier section is not completely accurate (and inconsistent with the picture presented there). The host doing the analysis must follow paths through the acknowledgment graph starting from a set of nodes representing messages sent by the host being checked. It cannot just check paths resulting from the last message received. The revised wording of Theorem 2 reads as follows:

Theorem 2

If there exists a path of positive acknowledgments or retransmissions to message Z from messages sent by host T and no negative acknowledgment has been issued for any message on the path by T or by any message acknowledged directly or indirectly by T then T has received message Z correctly. □

In order to construct a message reception algorithm based on Theorem 2, it is necessary that each host should construct an "acknowledgment graph" whose nodes are messages and whose arcs indicate acks, nacks, or retransmissions. A later section of this specification describes how the graph is constructed. The algorithm for analyzing the acknowledgment graph based on Theorem 2 is the following:

- Assume host S has sent message Z and requires confirmation that T received it.
- S must have observed message M_1, broadcast by T prior to the broadcast of Z.
- $M_2, \ldots, M_n$ are messages broadcast consecutively by T after M_1.
- Node S constructs the acknowledgment graph *starting with* M_2, and adding $M_3 \ldots$ incrementally.
- The leaves of the graph must be messages prior to Z.
- If any part of the graph cannot be constructed then it is undetermined whether the message Z has been received by the host T and the algorithm fails.
- If any one version of the graph satisfies Theorem 2, the message has been received.

The specifications of the transmission control and reception analysis algorithms of the TRANS protocol given above were found to require considerable development and interpretation during the implementation effort. The impleme..tation was finally based on the descriptions given above and a set of assumptions governing their interpretation. During the implementation, several additional problems with the specification were uncovered. Solutions for some of these problems were incorporated in the program, but others were discovered so recently that there was insufficient time to implement them.

6.2.1 Clarifications and Interpretations

The following clarifications and interpretations were developed during our prototype implementation and apply to the TRANS protocol as described above.

- There are two timeout values in the transmission control section of the protocol. If a host transmits a message and does not receive an acknowledgment of any kind within a certain time period then it will retransmit the message. This timeout will be called the *message timeout.* If a host creates a pending acknowledgment and does not have a genuine outgoing message to attach it to within a certain time period, then it will create a null message to carry the acknowledgments. This timeout will be called the *no-message timeout.*

- Unless stated otherwise, when the term "a message" is used in the transmission control section, it means any version of a message. A *message identifier*, however, is a particular [host name, sequence number, version number] triple. The distinction is between a conceptual message that is unchanged regardless of the version being considered and a particular transmission of a message. In the remainder of this report, the term *transmission* will be used to refer to a particular version of a message.

- Related to the previous item, if a host receives one version of a message uncorrupted then any other versions of that message that are received uncorrupted are not a new message.

- Acks and nacks must refer to specific versions (i.e., transmissions) of messages. To see this, suppose it were not so, and suppose that a host transmits a message X and later retransmits it in response to a nack. It is the sender's responsibility to keep retransmitting the message until the host that sent the nack receives it correctly. Accordingly, it must restart its message timeout timer and keep retransmitting until it receives an acknowledgment of some kind.. Now suppose a belated ack arrives for the *original* transmission. In the absence of version numbers, the sender might assume that this ack is acknowledging its retransmission and it will therefore turn off its message timer and stop further retransmissions—even though those who needed them have not acknowledged and may not have received them.

- Related to the item above: a host will not retransmit a message if the version number of the nack it receives is less than the version number of the most recent transmission of the message. This "old" nack can occur for a variety of reasons such as if the host missed the first transmission of the message that carried the nack or if the host that sent the nack missed a retransmission of the original message. The purpose of the nack was to cause a retransmission of the original message. This has already occurred and the normal operation of the protocol will cause the retransmission to be delivered to the host that issued the nack or another nack to be sent causing yet another retransmission. Responding to an "old" nack, however, can cause the unnecessary replay of a sequence of messages.

- A host will ignore a transmission of a message if it has previously seen that message uncorrupted. This implies that it will not examine the acknowledgments carried by the retransmission and it is for this reason that retransmissions must carry exactly the same acks and nacks as their originals. Another comment in the original specification of TRANS: "It is permissible, but not essential, for a node to broadcast a positive acknowledgment for a message that it had already received uncorrupted", does not appear meaningful because the transmission control rules do not allow a host to create two acks for the same message.

- A host is only required to ack a message once (directly or indirectly), not each version.

- The rule "when a node sees a positive acknowledgment for a message that it has not received, it adds a negative acknowledgment to its list" is interpreted to mean that 1) a host only needs to send one nack for a message that it has not received, not one for each version of the message, and 2) if a host has a pending nack for a message, then it does not need to add another if it sees an ack for a different version of the message. These assumptions follow from the previous assumptions about messages and transmissions.

- The statements "Node S constructs the acknowledgment graph starting with M_2, and adding M_3 ... incrementally" and "If any one version of the graph satisfies Theorem 2, the message has been received" imply that the following procedure is used for reception analysis. A group of nodes are added to the acknowledgment graph as a result of the first message transmitted by T after Z was transmitted, in this case M_2, and the transmissions that it acknowledges (directly or indirectly). The reception analysis algorithm is then applied. If it fails, then nodes are added as a result of the next message transmitted by T and the algorithm is applied again. This continues until the algorithm succeeds, the host runs out of messages, or part of the graph cannot be constructed.

- Messages from a host that follow a specific message M from that host do not affect the conclusions that can be drawn by analyzing the messages up to M from that host and the messages that they acknowledge. This is implied by the statement "if any one version of the graph satisfies Theorem 2, then the message has been received."

- The statement "If any part of the graph cannot be constructed then the algorithm fails" implies that analysis must stop if an acknowledgment is encountered for an unknown message (a message that the host performing the analysis has not seen) or if there is a gap in the sequence of messages from the host being checked.

- The sequence numbers issued by a host follow a regular pattern and are not just unique identifiers. Otherwise, a host would not be able to detect that there was a gap in the messages that it has received from another host and reception analysis would not be usable as described. This is based on the need to detect gaps mentioned in the previous assumption.

- The statement "S must have observed message M_1, broadcast by T prior to the broadcast of Z" refers to a message *actually seen* by S prior to broadcasting Z and not to the message that was actually last broadcast by T before S broadcast Z. S may not know the identity of the message that was actually last because it may have missed it due to an error. Similarly, M_2 is not necessarily the first message broadcast by T after S broadcast Z (it could have preceded Z). S knowing the identities of the messages from T whose broadcasts preceded and followed the broadcast of Z would improve the efficiency of the analysis but is not necessary.

6.2.2 Comments

This section contains comments about the TRANS protocol. Its purpose is to illuminate some of the properties of the protocol.

During reception analysis, a nack carried by a message acts as a barrier. No information along the path indicated by the nack could have been known to the host that issued the nack unless it can be reached by a different path. If the host knew about this other information, then the message carrying the nack would have also carried acks for these other transmissions.

When a nack is encountered during reception analysis, the host doing the analysis must assume that the host being checked did not know any information along the path indicated by the nack, even if this is not true. The host doing the analysis just cannot tell from the available information. The difference from the previous paragraph is that the message carrying the nack may not have been transmitted by the host being checked. It may just have been encountered along an acknowledgment path.

Nacks turn out to be self-healing. A nack will cause a new chain of acks to be started, which will eventually detour around the site of the error. So even if information indicating a valid reception is discarded when a host *issues* a nack, it will eventually be recovered due to the actions initiated by the nack.

When a host removes a pending ack for a message M because of a nack for M carried by another message, it agrees to wait and indirectly ack a future version of M. It will not create another pending ack for a version of M because it has already received an uncorrupted version of M. Removing the ack saves transmitting an acknowledgment, but it will probably increase reception latency.

With respect to transmission control, a nack for version v of a message indicates to the host that originated the message that a subset of the other hosts have not seen any version of the message up to v and that a new transmission is necessary. In effect, a new transmission due to a nack starts a new round of acknowledgment for the message with a smaller set of receivers. Hosts that have seen earlier versions of the message will ignore the new transmission, while the remaining hosts will try to ack it (directly or indirectly). (Note that, according to the previous paragraph, hosts that have seen the message before may still have to ack it indirectly.)

A host cannot remove a message until it has been received by all other hosts. Initially, it might appear that a message can be unremovable for an indeterminant amount of time because some host might not be broadcasting messages or might be sending messages without acknowledgments. In practice, neither of these conditions can occur for an appreciable length of time. If they do persist, then there is a serious problem with the host or the network that should be handled by other mechanisms. Note that determining that a message has been received by a particular host can take an indeterminant amount of time, but it would be for different reasons such as an unfortunate sequence of errors that cause key messages to be missed.

Every host must broadcast a message in a bounded amount of time unless there is a serious problem in the network. This time period depends on the values of the protocol timeouts and the number of errors that will be tolerated until a network problem is considered to exist. Consider a host H. Assume that H was the last host to broadcast a message. If no acknowledgment for the message is received within a message timeout period, then H will retransmit the message. If a nack is received then H will automatically retransmit the message. If an ack is received then H will create a pending ack for the new message. If a client message is not received within a no-message timeout period, then H will create a null message to carry the pending ack. Now assume that H was not the last host to broadcast a message. If H saw the message then it has a pending acknowledgment for it. If a client message is not received within a no-message timeout period, then H will create a null message to carry the pending acknowledgment. If H did not see the message then the original host will retransmit the message after a message timeout period because no host responded with an acknowledgment or another host will send a message acknowledging the first message. This new message will now be the last message and H will respond to it in the same way. If H does not respond to some number of messages in a row then there is a serious problem in the system.

Once a host has a pending acknowledgment it cannot simply remove that acknowledgment because of information seen in a new message. Instead, it must replace that acknowledgment with a different one depending on the contents of the new message. Thus, once a host creates a pending acknowledgment, it must send a message and that message must carry an acknowledgment. If no client message arrives it must create a null message when the no-message timer goes off.

The TRANS protocol causes a never-ending sequence of messages to occur. Even if no client messages are being received anywhere, the protocol will continue sending null messages or retransmissions to respond to the last message sent. Usually, there will be a lot of activity in the system because many hosts will be broadcasting client messages. This helps overcome the effect of a few errors.

Each message sent by a host must in general carry an acknowledgment for at least one other message. This acknowledgment will then indirectly acknowledge a series of other messages. The main reason that there must be an acknowledgment is that once a host has a pending acknowledgment it must send it or replace it with another acknowledgment, as discussed above.

It is possible for a message to be sent without any acknowledgments. This can occur if a host sends two messages before another host has sent any. Since the first message carried all of the host's pending acknowledgments, the second will not carry any. In some sense, the second message is implicitly acknowledging the first message. Another way this can occur is if a host issues a message after missing, due to errors, all messages that were sent from the time it sent its previous message. Once again, the first message will carry all of the pending acknowledgments and the second will not carry any. Neither of these situations can occur for long unless the network is having serious problems. (The first message transmitted also does not carry any acknowledgments.)

6.2.3 Problems

Several problems with the TRANS specification were uncovered during the implementation. This section identifies problems that were addressed in the implementation.

- The specification indicates that a message will be retransmitted if no acknowledgment is received for that message before a message time-out occurs. Presumably additional retransmissions will be made if additional message time out periods elapse without an acknowledgment being received. The protocol does not specify what happens if the timeout is satisfied by receipt of an acknowledgment, but then a retransmission occurs because of a nack. Should a new timeout period be started or is it unnecessary? It turns out that a new timeout period must be started whenever a new version of a message is transmitted.

This can be seen from the following counter-example. Let there be three hosts A, B, and C. A sends a message W that is seen by B and missed by C. B sends a message X that carries an ack for W. A and C see X. A turns off its message timer for W. C sends a message Y that carries an ack for X and a nack for W. A retransmits W as W′. B ignores W′ and C misses it. A now sends Z which carries an ack for Y. C sees Z and turns off its message timer for Y. At this point, C has not seen W and W can no longer be retransmitted. It will not be automatically sent because its message timer is off. The one message that carries a nack for W, Y, will also no longer be retransmitted. Y's message timer is also off, and all other hosts saw Y so a nack will never be issued for it.

- According to the previous item, there is a sequence of timeout periods for a message, some started because no acknowledgments have been received and some started because of nacks. As was mentioned in the comment section above, each retransmission issued because of a nack starts a new round of acknowledgments for a message. A problem will occur if a message timer is turned off because of an ack for an earlier round. That is, if the last round was started because of a nack for version v of a message then the message timer cannot be turned off for an ack for a version w of the message where $w \leq v$. An "old" ack for some message M can occur if the host that issued M missed previous transmissions of the message carrying the ack and is now seeing it for the first time.

  The problem that occurs is that the message may not have been seen by some hosts but will not be retransmitted because of the message timer or a nack. This can be seen in the following scenario. Let there be three hosts A, B, and C. A sends a message W that is seen by B and missed by C. B sends a message X that carries an ack for W. X is seen by C and missed by A. C sends a message Y that carries an ack for X and a nack for W. Y is seen by A and missed by B. A retransmits W as W′. B ignores W′ and C misses it. Now the message timer for X goes off and B retransmits it as B′. B′ is seen by A and ignored by C. A now sees the ack for W and turns off W's message timer. At this point, C has not seen any version of W and W can no longer be retransmitted. It will not be automatically sent because its message timer is off. In addition, the only nack for W is for an earlier version and will be ignored.

- The specification given earlier states: "Each message carries with it, one or more acknowledgments to previous messages." This is not true as was discussed in the comments section above. Later, in the reception analysis section, it states that "The leaves of the graph must be messages prior to Z." This is also not true for the same reasons.

- Theorem 2 contains the clause "and no negative acknowledgment has been issued for any message on the path by T or by any message acknowledged directly or indirectly by T." Initially, it was assumed that the phrase "or by any message acknowledged directly or indirectly by T" meant that there would be a chain of positive acknowledgments leading from a message issued by T to a message that contained a negative acknowledgment. It turns out that this is insufficient and that some negative acknowledgments can be missed, allowing message reception to be falsely detected. The phrase should read "or by any message acknowledged directly or indirectly by T with a chain of positive or negative acknowledgments starting with a positive acknowledgment."

  This problem can be seen in the example shown in Figure 6.3. Assume the messages shown in the Figure were each sent by a different host and Let $H(x)$ denote the host that sent message x. Given the information in the Figure, can H(Z) deduce that H(U) saw Z? (Assume that H(X) did not see Y and that H(Y) did not see X when X and Y were transmitted.) According to the discussion above, H(Z) would not be able to conclude that H(U) had seen Z. Although there is a positive acknowledgment chain $U \rightarrow W \rightarrow X \rightarrow Z$ from U to Z, it is

```
              Z
    nack   / \   ack
          /   \
         Y     X
    nack  \   /  ack
           \ /
            W
            |  ack
            |
            U
```

Figure 6.3: Reception Analysis must start from an ack

negated by the acknowledgment chain $U \rightarrow W \not\rightarrow Y \not\rightarrow Z$ (where $\rightarrow$ and $\not\rightarrow$ indicate acks and nacks, respectively.) As shown in the following discussion, H(Z) can plausibly deduce both that H(U) has seen and not seen Z. It must assume the worst and assume that *H(U)* has not received Z.

Case 1. Assume that H(U) has seen Z.
Assume that H(U) first sees Y. It will replace its ack for Z with an ack for Y. Assume that H(U) then sees X. It will then create an ack for X. When H(U) sees W, it will replace the acks for X and Y with an ack for W. U can then be issued with only an ack for W.

Case 2. Assume that H(U) has not seen Z.
Assume that H(U) first sees X. It will then create an ack for X, and a nack for Z. Assume that H(U) then sees Y. It will then replace its nack for Z with an ack for Y. When H(U) sees W, it will replace the acks for X and Y with an ack for W. U can then be issued with only an ack for W.

- Theorem 2 gives contradictory information about the effect of a nack on a path of positive acknowledgments or retransmissions. In the example shown in Figure 6.4, the positive acknowledgment path $W \rightarrow X \rightarrow Z$

```
              Z
     nack   / \   ack
           /   \
          Y     X
      ack \     / ack
           \   /
             W
```

Figure 6.4: Positive Acknowledgment Path Invalidated by nack

should be invalidated by the acknowledgment path $W \rightarrow Y \not\rightarrow Z$. However, in the two examples shown in Figure 6.5, the positive acknowledgment paths $W \rightarrow Z' \rightsquigarrow Z$ and $U \rightarrow Z' \rightsquigarrow Z$ (where $\rightsquigarrow$ indicates retransmission) should not be invalidated by the acknowledgment paths $W \rightarrow Y \not\rightarrow Z$. In the second (rightmost) example of Figure 6.5 W and X are issued by the same host, with W preceding X. This problem could

```
        retransmit                 retransmit
        Z--------Z'                Z-------Z'
  nack  / \  ack  |          nack  |       |
       /   \      |                |       |
      Y     X    /                 Y       | ack
   ack \   / ack / ack         ack |       |
        \ /     /                ..|.......|..
        W------                  : W       U :
                                 :...........:
```

Figure 6.5: Positive Acknowledgment Paths *not* Invalidated by nack

be solved by carefully rewording the theorem, but it is not clear why the path must return to the initial version of Z. Another approach is to drop retransmission arcs from the graph and reword the theorem to read "If there exists a path of positive acknowledgments to message Z or one of its retransmissions from messages sent by node T ... has received message Z correctly." A retransmission of a message carries exactly the same acknowledgments as the original message so it can be used directly to continue analysis of the graph.

6.3 Implementation

The protocol was implemented on a network of Sun workstations connected by an Ethernet. Two programs were written, one for the protocol and one for a driver used to exercise the protocol. The programs were written in C and run under the UNIX operating system. Each workstation contains a protocol and a driver process. The driver periodically sends a message to its protocol process, which then broadcasts the message to other protocol processes. The protocol program consists of two main sections: one for transmission control and the other for reception analysis. Initial performance measurements were made with two host sets and four error levels.

The program is written modularly so that alternate protocol rules can be examined. In particular, the reception analysis section is completely separate from the rest of the program and can be replaced by more efficient versions in the future. There are a variety of program options that can be set when the protocol programs are started. This permits a wide variety of configurations to be set up for experimentation.

Reception analysis could not be implemented directly from the specification given in the original description of TRANS. As explained earlier, that description mixes existence-proof arguments with implementation directions, and does not describe an efficient algorithm for building, maintaining, and examining the acknowledgment graph. In addition, certain implementation details were missing—such as an indication of when graph information can be discarded. It was decided to maintain an up-to-date graph and to add information as transmissions arrived. When information was added to the graph, the appropriate analysis would be performed to determine whether new receptions could be confirmed. Information would be removed from the graph as soon as it was no longer needed. Specific version numbers must be indicated for information in the acknowledgment graph.

6.3.1 Top-Level Design

A prototype version of the TRANS protocol was implemented to run on a network of Sun workstations connected by an Ethernet. The workstations were running the Sun 3.4 version of the UNIX operating system. Although TRANS is a link-level protocol, it was implemented at the presentation layer. This greatly simplified the implementation while still allowing a realistic and thorough evaluation.

The protocol was written in the C programming language and was run as a single process. A second program called the driver was also written in C and was also run as a single process. Each workstation contained a single protocol process and a single driver process. A driver represented the set of clients on a workstation using TRANS, and sent broadcast messages to the protocol process on its workstation. The drivers sent messages with a Poisson inter-arrival rate. When a protocol process received a message from its driver, it broadcast the message to the other protocol processes.

Communication between the processes was accomplished with the User Datagram Protocol (UDP). Each protocol process was connected to its driver through a port bound to the host's address and the set of protocol processes were connected through a port bound to the broadcast address. Messages were broadcast among the protocol processes with the UDP broadcast mechanism. No errors were simulated for communication between a driver and its protocol process. An error rate could be individually set for each protocol process, however, to control message reception from other protocol processes.

6.3.2 Implementation Decisions

The following decisions were made when the protocol was implemented:

- It would not be possible for a host to receive a message with a corrupted body and an uncorrupted header. A message would either be fully received or not received at all.
- The length of the no-message timeout would be the same for pending acks and nacks.
- A list of old nacks would not be maintained. An old nack refers to a transmiss. ı for which the current host previously issued a nack.

As a resuit, a host can issue multiple (redundant) nacks for the same transmission. (A host must issue a new nack for a new version of a message that it has not seen to indicate that it missed that retransmission too.)

6.3.3 Data Structures

Transmission Control

The data structures for transmission control are straightforward. A list for pending acks and a list for pending nacks must be maintained until a message is sent and these lists can be cleared. A list of pending messages must also be maintained. These are messages that have been sent by the host but have not yet been verified as received by all other hosts. One additional list is maintained of old acks. This list represents messages that have been seen by the host in an uncorrupted form. Any message the host has seen, whether or not the host actually issued an ack for it, is represented in this list (except those still in the pending ack list).

The old-ack list is actually represented in a condensed form. There is a separate number/linked-list pair for each host. The number is the largest message sequence number from a host that the current host has seen such that all preceding messages from that host have been seen. The linked list, which is maintained in ascending order, keeps track of more recent messages. The first node in the linked list represents a missing message and the last node in the linked list represents the most recent message seen from that host. As gaps are filled in, the linked list is condensed and the latest consecutive sequence number is raised. A more sophisticated data structure will be needed if sequence numbers can be reused.

Acknowledgment Graph

Several data structures are used to represent the acknowledgment graph. The main data structure is the node list. It represent[illegible] aph of all known messages that may be needed to help detect [illegible]ception of a host's messages. Another structure, called the host list, keeps track of the messages that have been seen from each host. It is not required, but it simplifies examination of the node list.

The node list maintains an up-to-date representation of the pertinent message traffic. Each node represents a single message. When a new message is sent or a new, relevant message arrives, a new node is added to the list. A special node, called an *unknown node*, must also be added when a message that has not been seen before is detected as an acknowledgment in another message. Unknown nodes are filled in when their corresponding messages are seen. Nodes are removed from the node list when they are no longer needed in the message graph. Nodes that represent a host's own messages must also be kept until that message has been verified as received by all other hosts.

Each node has two linked lists, one for the acks carried by the message and one for the nacks. Acknowledgments are only added to one of these lists if they are needed in the graph. An acknowledgment is not needed if it refers to a message that has already been seen but is no longer represented by a node in the graph. These linked lists contain full message identifiers, not pointers to nodes.

The host list is similar to the old-ack list used for transmission control. There is a separate number/linked-list pair for each host. The number is the largest message sequence number from a host that the current host has seen such that all preceding messages from that host have been seen and removed from the node list. The linked list, which is maintained in ascending order, keeps track of more recent messages. A more recent message can be in one of three states:

gap: not received or referred to by an acknowledgment.

gone: received and no longer represented in the node list.

node: received or referred to by an acknowledgment and still represented in the node list (including "unknown" nodes).

The first message in the linked list is either "gap" or "node", and the last message in the list is either "gone" or "node". As messages at the beginning of the list become "gone", the list is condensed and the latest consecutive sequence number is raised. A more sophisticated data structure will be needed if sequence numbers can be reused. Once again, the linked lists contain full message identifiers, not pointers to nodes.

Reception Analysis

Message reception analysis is performed with the data structures that form the acknowledgment graph. A matrix is used to keep track of which messages have been received by which hosts. The columns of the matrix are the hosts in the system and the rows are the pending messages.

6.3.4 Algorithms

Transmission Control

The algorithms for transmission control follow directly from the rules. They control what happens when a message arrives from a client or another host or a timer goes off. The important transmission control algorithms are:

```
New Client Message Arrives
    increment sequence number
    add pending acknowledgments to message
    broadcast message
    start message timer
    turn off no-message timer (if necessary)
    make a node for the message
    add the node to the graph

New Broadcast Message Arrives
    create a pending ack for the message
    start no-message timer (if necessary)
    process message ack list
    process message nack list
    make a node for the message
    if node is needed
        add node to graph

Process Message Ack List
    if ack for my message
        if for latest round for message
            turn off message timer
    else if acked message is unknown
        create nack for unknown message
        start no-message timer (if necessary)
    else if pending ack exists for message
        remove pending ack
        cancel no-message timer (if necessary)

Process Message Nack List
    if nack for my message
        if for most recent version of message
            start new round for message
            rebroadcast message
            restart message timer
    else if pending nack exists for message
        remove pending nack
        cancel no-message timer (if necessary)
    else if pending ack exists for message
        remove pending ack
        cancel no-message timer (if necessary)

Message Timer Goes Off
    rebroadcast message
    restart message timer

No-message Timer Goes Off
    increment sequence number
    create null message
    add pending acknowledgments to message
    broadcast message
    start message timer
    make a node for the message
    add the node to the graph
```

Acknowledgment Graph

The acknowledgment graph algorithms are use to add nodes to and remove nodes from the graph. When a node is made for a message, acknowledgments are added only if they refer to other graph nodes. An important task of the algorithms is creating and filling in unknown nodes. When an unknown node is filled in, all hosts are checked to see if any new message receptions can now be detected—the unknown node(s) may have been blocking many paths. When a new message is added, only the host that sent that message must be checked. When it is determined that a message has been received by all other hosts, an attempt is made to remove the node for that message from the graph. Removal of a node can cause a sequence of node removals to occur. The important algorithms are:

```
Make a Node for a Message
    for each (n)ack in the message (n)ack list
        if (n)acked message is unknown
            create unknown node for (n)acked message
            add unknown node to graph
            add reference for unknown node in host list
            add (n)ack to node (n)ack list
        else if (n)acked message has a node in the graph
            add (n)ack to node (n)ack list

Add a Node to the Graph
    if node was unknown node in graph
        fill in all copies of unknown node
        if new node is for a new version of message
            add node to node list
        check all hosts for new reception
    else
        add node to node list
        add reference for node in host list
        check host for new reception

Remove Nodes from the Graph
    repeat
        look for a node that
            1. has empty ack and nack lists, and
            2. if it is my message, has been received by all hosts
        if found
            remove node from node list
            remove all references to node from other node's ack and nack lists
            mark as "gone" in host list
```

```
until no such node is found
```

Reception Analysis

Detecting whether another host has seen any of a specific host's messages is accomplished by examining the acknowledgment graph in a specific order. The search starts with the first message from the other host that has not been marked as "gone." It then continues sequentially through the remaining messages known to have been sent by that host. If a gap is found in the node's message sequence or an unknown node is found in the graph, then the search is aborted. This corresponds to detecting that a portion of the graph cannot be constructed in the reception analysis specification. Any receptions that have been detected up until the search is aborted are valid.

The graph is searched in two phases. First, all paths leading from the other host's current message are examined for unknown nodes and *bad* nodes. Bad nodes are nodes that cannot be used in a positive acknowledgment path because they are nacked by some acknowledgment path. The list of bad nodes continues to grow as the analysis proceeds through the other host's messages. If an unknown node is found the search is aborted. The second phase examines all positive acknowledgment path leading from the other host's current message. No unknown nodes should be found now because they would have been detected in the first phase. A positive acknowledgment path is abandoned if it leads to a bad node. When one of the messages from the host doing the analysis is found, it is marked as received.

The important algorithms for reception analysis are:

Check All Hosts

```
for all hosts (except mine)
    check host
```

Check Host

```
start with first non-"gone" message for host being checked
while not "gap" message
    if node is unknown
        break
    if any nack's in node's nack list are unknown
        break
        add nacks in node's nack list to bad list
        find bad nodes starting from node's ack list
        if unknown node found while searching for bad nodes
            break
        search for received messages from node's ack list
        move to next message for host being checked
    if any of my messages were received by all hosts
        remove nodes
```

Find Bad Nodes

```
    for each acknowledgment on list being checked
        if unknown node
            break
        if one of my messages
            ignore (this earlier message has already been checked)
        else find bad nodes starting from node's ack list
        if unknown node found while searching for bad nodes
            break
        find bad nodes starting from node's nack list
        if unknown node found while searching for bad nodes
            break
```

Search for Received Messages

```
    for each ack on node's ack list
        if in bad list
            ignore ack
        else if one of my messages
            mark as received
            if received by all other hosts
                set received-by-all flag
        else search for received messages from node's ack list
```

6.4 Performance Measurements

Several preliminary performance measurements were taken to obtain a general understanding of the behavior of the TRANS protocol. Our main interests were determining how much storage would be required for the acknowledgment graph and for reception analysis, how long it would take for a host to detect that one of its messages had been received by all other hosts, how long the pending message queue would grow, and how many extra messages would be required for reliable delivery. We also wanted to see how these values would change as the error rate or the number of hosts in the network grew.

The system can be configured by setting the number of hosts in the network, and for each host 1) the message reception error rate, 2) the message timeout value, 3) the no-message timeout value, and 4) the average client message inter-arrival rate. The number of hosts was set at four and eight. The error rate was the same for all hosts during a trial. With four hosts, it was set at 0%, 1%, 5%, and 10%. With eight hosts, it was set at 0% and 1%. Relatively low error rates were used because the protocol is really intended for networks with very reliable basic communication media such as an Ethernet. The no-message timeout value was set at 1.5 times the average inter-arrival rate. This ensured that there would be times when the no-message timer would go off. The message timeout value was set at twice the no-message timeout value. The message timeout value should probably be set to be greater than the no-message timeout value so that hosts have a chance to return acks before the original message is rebroadcast. The timeout values and the inter-arrival rate were set the same for all hosts, at 18, 36, and 12 seconds respectively.

The results from the trials are shown in Tables 6.1 and 6.2.

The performance figures obtained were reassuring and show that TRANS does perform well. There were very few surprises, with most values rising as the error rate grew or the number of hosts grew. The extra messages that were sent were divided into those attributed to nacks, message timeouts, and no-message timeouts. Pending messages were messages that had been sent by a host but not yet detected as received by all other hosts. Latency indicates the time between when a message is sent and when it is detected as received by all other hosts. "Latency to remove" is the time between when a message is sent and when its related information is actually discarded. This would be expected to be longer that the simple latency time because graph information sometimes needs to be kept in order to resolve paths to other nodes.

The percentage of extra messages grew as the error rate increased and the number of hosts increased. It was always very low however and com-

Number of hosts = 4	error rates			
	0%	1%	5%	10%
# of client messages	308	308	308	307
# of messages sent	367	383	428	498
# of extra messages sent	59	75	120	191
% extra messages	19	24	39	62
# extra messages due to:				
nack	0	11	50	101
message timeout	0	0	5	22
no-message timeout	59	64	65	66
max. pending messages	8	8	11	17
ave. pending messages	1.25	1.40	2.02	3.20
max. nodes in graph	12	33	44	83
ave. nodes in graph	2.46	3.00	5.56	11.17
latency times in seconds:				
max. latency	19	89	102	172
ave. latency	12.29	13.36	18.00	26.54
max. latency to remove	19	89	102	172
ave. latency to remove	12.[illegible]	13.56	19.61	30.62

Table 6.1: Performance Measurements with 4 Hosts

Numbe. of hosts = 8	error rates	
	0%	1%
# of client messages	303	303
# of messages sent	378	403
# of extra messages sent	75	100
% extra messages	25	33
# extra messages due to:		
nack	0	26
message timeout	0	.5
no-message timeout	74	74
max. pending messages	8	8
ave. pending messages	1.56	1.79
max. nodes in graph	23	64
ave. nodes in graph	6.72	8.91
latency times in seconds:		
max. latency	19	67
ave. latency	14.94	16.43
max. latency to remove	19	67
ave. latency to remove	14.96	17.12

Table 6.2: Performance Measurements with 8 Hosts

pares very favorably with point to point protocols that would require $O(N)$ messages for every message sent. Here, the number of extra messages was $O(1)$. Even more favorable results could have been obtained by increasing the no-message timeout with respect to the inter-arrival rate so that client messages could carry more pending acknowledgments. This would possibly cause more message timeouts, but they seem rare anyway.

The number of message timeouts increased as the error rate increased. This implies that a few messages carrying key acknowledgments were dropped, thereby forcing hosts to retransmit some messages.

The number of pending messages grew with the error rate but the average number was very small. Similarly, the average number of nodes required in the acknowledgment graph was pleasingly small, even with an error rate of 10%. It was encouraging to see that the amount of storage required was not prohibitive.

The latency times were also very respectable and were close to the client message inter-arrival time. The other latency time of interest is the time between when a message is sent and when it is *actually* received by all other hosts. This value was not measured but it is expected to be very small. It interesting to note that the removal latency time is very close to the simple latency time. This indicates that when a message is detected as received by all hosts its related information is basically ready for removal. This helps prevent large storage requirements.

6.5 Problems Discovered

Several problems and suspected problems were discovered in our implementation of the TRANS protocol. Further analysis may determine that the suspected problems are actually handled correctly, but there was insufficient time to find solutions to the known problems or examine the suspected problems. It is believed that the problems are not too deep and can be solved.

TRANS seems to work well in ideal situations where there are few errors, all hosts see all messages in order, and actions occur instantaneously. The problems with the protocol that were described earlier and in this section arise from several sources:

- Because of errors, it is possible for a host to see messages out of order. This means that acknowledgments for later versions of messages can be seen before acknowledgments for earlier versions, later versions of messages can be seen before acknowledgments for earlier versions, later messages from a host can be seen before any version of an earlier message from that host, and so on. The effect is that actions can be required before a host knows the full message history.

- Version numbers are important in certain situations. It is not always sufficient to consider which message is involved before taking an action, sometimes the specific transmission must be considered. For example, an ack for an earlier version of a message cannot turn off a message timer for a later round.

- Reception analysis seems to require more information to work properly than transmission control. Both use information acquired from acknowledgments in messages to perform their duties, but it seems that the acknowledgments that are sufficient to perform transmission control are insufficient to perform reception analysis. More study is required to determine whether transmission control needs to be redesigned to include more acknowledgments or reception analysis needs to be redesigned to make better use of the information that is currently available.

- Multiple acknowledgments can be received for the same message. This can occur because errors keep one host from seeing another host's message or because the network is not ideal and two hosts can act simultaneously.

The following problems have been identified.

- The way that message timers are handled is still not correct. The current approach will not work if the following occurs. Assume that the latest round for a message from host H starts with version 1 and that H has also transmitted versions 2 and 3. Because of errors, H has not seen message M which carries an ack for version 1 or message N which carries a nack for version 2. M now arrives and H turns off the message's timer. Now N arrives and is ignored because it nacks an old version of the message. The timer for M will not be turned on again and some host has still not seen M. The rules for determining when a message timer can be turned off, an ack or nack for an old version of a message can be ignored, a message can be removed, and a message should be rebroadcast must be reexamined.

- A host must form a pending nack for every version of a message that it sees acked in another message. This contradicts the rule that says that only one pending nack must be maintained for a message. In fact, a pending nack must be maintained for each transmission of a message. This can be seen in the following situation (see Figure 6.6). Assume that there are four hosts A, B, C, and D. A broadcasts mes-

```
        Z         Z'
   ack  |\ nack   | ack
        | \       |
        |  \      |
        Y   |     X
   ack   \  | ___/  ack
          \|/
           W
```

Figure 6.6: Pending nack must be Maintained for each Retransmission

  sage Z which is seen by B and missed by C and D. Before B can respond, Z's message timer goes off and A retransmits Z as Z'. Z' is ignored by B, seen by C and missed by D. Now B transmits message Y which carries an ack for Z. It is seen by A and D but missed by C. D forms a pending nack for Z. C now transmits message X which carries an ack for Z'. X is seen by everyone. D does not form a pending nack for Z' because it already has one for Z. Now D issues message W which carries a nack for Z. The current situation is shown in Figure 6.6. If A now performs reception analysis, it would incorrectly determine that D has received Z'. This is an example of a situation where reception analysis requires more information than transmission control.

- A similar situation occurs when a host H has a pending nack for message M and sees a message N which carries a nack for M. According to the transmission control rules, H should replace its pending nack with an ack for N. Actually, it should just add a pending ack for N if its nack is for a different version of M than the nack carried by N. Otherwise, reception analysis would incorrectly find a reception path. This situation is shown in Figure 6.7. Assuming that the host

```
      Z     Z'
ack   |     |  nack
      |     |
      Y     X
ack    \   /   ack
        \ /
         W
```

Figure 6.7: Retransmissions Require Individual nacks

that issued W did not see Z, message W should carry a nack for Z and shouldn't have removed it because of the nack for Z′ carried by message X.

- Similar questions arise about other rules for replacing acknowledgments. For example, it appears likely that older acknowledgments for a message should not replace newer acknowledgments for the message.

6.6 Conclusions and Recommendations for Future Work

Our prototype implementation of the TRANS broadcast protocol was undertaken very near the end of the contract with only very limited time and money available. Unfortunately, it did not prove possible to implement the protocol, solve its outstanding problems, and undertake an extensive performance evaluation within the resources available. We chose to concentrate on completing the prototype implementation, identifying problems, and obtaining preliminary performance measurements.

Our initial measurements on the performance of TRANS have been favorable. Its storage requirements, network bandwidth usage, and reception detection latency were all found to be quite low. It appears that TRANS would make an excellent foundation for a variety of protocols and distributed systems algorithms for broadcast environments.

Our prototype implementation of TRANS provides an excellent test-bed environment for further work on this and related broadcast protocols. Some recommendations for future work that could build on the our achievement so far are presented below.

6.6.1 Corrections and Formal Specification

A great deal was learned about the behavior of TRANS during this investigation and several subtle problems were uncovered. We do not consider the remaining problems to pose significant difficulties, but simply had insufficient time to resolve them. Further investigation and development of TRANS must begin with the correction of the problems already discovered. Because of the subtlety of the issues involved, it is important for a corrected version of TRANS to be specified formally and completely, and subject to formal analysis and proof. There are at least three properties of TRANS that should be specified and formally verified:

Liveness of Transmission Control: we need to be sure that a message will be rebroadcast until all hosts have received it.

Safety of Reception Analysis: we need to be sure that when the reception analysis algorithm declares that a particular host has seen a particular message, then indeed it has seen that message.

Liveness of Reception Analysis: we need to be su [illegible] t if a host has received a message, then eventually the reception analysis algorithm will enable the sender of the message to discover that fact.

The proof given for Theorem 2 in Part III of this report is a proof of the second of these properties. Although valid, it is deficient in that it is conservative: it does not address the issues surrounding retransmissions and multiple versions. A complete formal analysis of TRANS would be an extremely challenging and worthwhile exercise, since the protocol is one of the most subtle distributed algorithms we have encountered.

6.6.2 Additional Performance Measurements

In the time available, we were able to perform only very limited performance measurements on our implementation of TRANS. Additional measurement and performance evaluation is clearly necessary in order to determine the practical utility and characteristics of TRANS. Among the performance properties that should be investigated we suggest the following as particularly relevant:

- Compare against point to point and existing protocols for broadcast communications.
- Vary more parameters, and vary parameters over a wider range. Parameters include message timeout, no-message timeout, error rates, number of hosts, and their communication patterns (e.g., equally active hosts, one major sender with the rest mainly passive). Observe the change in performance characteristics of the protocol as these parameters change, and check that it is robust.

6.6.3 Performance Improvements

Although TRANS performed well in the initial measurements, there are several alternative versions that appear to offer better performance or different trade-offs. There are many ways to measure a protocol, such as the number of packets sent, the total number of bits sent, the percentage of overhead information sent, the amount of state that must be kept, the amount of processing required, the time until all hosts have received a message, the time until a sender realizes that all hosts have received one of its messages, etc. There are tradeoffs between these properties that should be evaluated for different environments and requirements.

TRANS is optimized to reduce network traffic, especially the number of acknowledgments that need to be sent. This is accomplished by establishing acknowledgment chains and reducing the number of explicit acknowledgments to a minimum. The problem is that these acknowledgment chains become very tenuous and contain very little redundant information. As the error rate increases, it can be very difficult for a host to determine that other hosts have received particular messages. This can cause so many retransmissions that the initial network traffic savings will be negated. TRANS tends to favor low network traffic over low latency, storage, and processing times. As the error rate increases, there is the danger of significant increases in storage, processing, and reception detection latency times. In general, it appears that minimizing the number of acknowledgments is not always the best choice.

Small changes in the protocol can address these problems. A receiver always knows exactly what it has seen whereas it may be hard for a sender or a third party to determine whether that receiver has received a particular message. The TRANS reception analysis algorithm must err on the side of caution—it must fail to conclude that a message has been received if there is any doubt that it has. We believe that considerable improvements in some performance characteristics can be obtained by having receivers send a few redundant acks and nacks in order to resolve ambiguities in a sender's reception analysis. For example, a *direct* ack can always be believed, even if other acknowledgment paths to the same message contain nacks. Thus a host that *has* received a message that it sees others nacking can unambiguously affirm its reception of the message by appending its ack directly to one of its own messages, rather than relying on transitivity.

6.6.4 Extensions to Functionality

Our implementation of TRANS provides for broadcast communication using a physically broadcast medium. Extensions to the functionality of the protocol could include multicast and the extension to bridged collections of broadcast networks where only a subset of hosts sees each broadcast.

6.6.5 Use of Broadcast Communications in Distributed Algorithms

Mutual exclusion, locking, synchronization, and distributed database update algorithms provide good examples of applications in which broadcast communication can provide substantial benefits.

Consider, for example, a tactical environment comprising multiple sensors, databases, and actuators using broadcast communications. As each sensor broadcasts its readings, those databases that hear the broadcast will update their records. When an actuator subsequently broadcasts a request for a value, any database that hears the request can broadcast a value in reply. Other databases will see this request-response and can use it to update their own records of the value concerned (if they missed the latest sensor broadcast), or can override it with a broadcast of their own if they see that the first response produced an obsolete value. In this way, replication and consistency of the databases is achieved in a very robust manner, with very little message overhead, and very little explicit coordination. Of course, the metric that determines which values are more desirable need not be based simply on time (where newer values are preferred), but could consider accuracy or other properties of the data.

6.6.6 Concluding Remarks

As far as we are aware, TRANS is the first protocol that exploits the characteristics of broadcast communications in order to achieve more than simple broadcasts. TRANS uses the fact that all parties see the traffic of all others to significantly reduce the number of explicit acknowledgments that are required. The price paid is in the latency of confirmed message reception, in the complexity of the reception analysis algorithm, and in the space required to store information needed by that algorithm. Our prototype implementation indicates that these costs are not excessive and that protocols of this kind should be viable in practice.

Our implementation revealed some problems with the protocol as it stands at present. In the time available we were unable to correct all the problems encountered and were also unable to collect all the performance data required for a full evaluation. Given our prototype implementation, it would require relatively little additional work to remedy the outstanding problems and perform a substantial performance evaluation. We have identified simple modifications to the TRANS protocol that could significantly improve some of its performance characteristics and we have identified situations (such as the management of replicated databases) in which the use of broadcast communications could support new algorithms for the coordination of distributed systems. Our prototype implementation of TRANS provides an excellent test-bed environment for further work on this and related broadcast protocols and algorithms.

Bibliography

[1] P. M. Merlin and B. Randell, "State Restoration in Distributed Systems," *IEEE Fault Tolerant Computing Symposium-8*, 1978, pp. 129-134.

[2] B. Liskov *et al.*, "Orphan Detection," *IEEE Fault Tolerant Computing Symposium-17*, 1987.

[3] J. Chang and N. F. Maxemchuk, "Reliable Broadcast Protocols," *ACM Transactions on Computer Systems*, Vol. 2, No. 3, August 1984, pp. 251-273.

[4] F. Schneider *et al.*, "Reliable Broadcast Protocols," *Science of Computer Programming*, Vol. 3, No. 2, March 1984.

[5] P. V. Mockapetris, "Analysis of Reliable Multicast Algorithms for Local Networks," *ACM Eighth Data Communications Symposium*, 1983, pp. 150-157.

[6] R. Gueth *et al.*, "Broadcasting Source Adressed Messages," *IEEE 5th International Conference on Distributed Computer Systems*, 1985, pp. 108-115.

[7] F. Cristian, "High Availability of Computer Service despite Component Failure," Canadian Annual Information Processing Symposium, 1987.

[8] G. Ricart and A. K. Agrawala, "An Optimal Algorithm for Mutual Exclusion in Computer Networks," *Communications of the ACM*, Vol. 24, No. 1, January 1981, pp 9-17.

[9] L. Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, Vol. 21, No. 7, July 1978, pp. 558-565.

[10] R. L. Schwartz and P. M. Melliar-Smith, "From State Machines to Temporal Logic: Specification Methods for Protocol Standards," *IEEE Transactions on Communications*, Vol. COM-30, No. 12, December 1982, pp. 2486-2496.

MISSION
of
Rome Air Development Center

RADC plans and executes research, development, test and selected acquisition programs in support of Command, Control, Communications and Intelligence (C^3I) activities. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C^3I systems. The areas of technical competence include communications, command and control, battle management information processing, surveillance sensors, intelligence data collection and handling, solid state sciences, electromagnetics, and propagation, and electronic reliability/maintainability and compatibility.